INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA DE INTELIGÊNCIA ARTIFICIAL

nodeML - Towards reproducible ML in federated environments

EDGAR SIMÃO DA MOTA E SILVA
outubro de 2022

nodeML
Towards reproducible ML in federated environments

Edgar Simão da Mota e Silva

Student No.: 1161504

**A dissertation to obtain the degree of
Master in Artificial Intelligence Engineering**

Supervisor: Doutor Luiz Felipe Rocha de Faria, Professor Coordenador do Instituto Superior de Engenharia do Instituto Politécnico do Porto

Porto, October 26, 2022

Dedicatory

To my parents, who gave everything for my personal and professional life, and my girlfriend, without whom I would have never been able to make it to this point.

"Luta e terás, pondera e serás" - António Silva

Abstract

Advances and increasing interest in AI (Artificial Intelligence) in the field of health have created novel issues, namely explainability and reproducibility of ML (Machine Learning) models.

In addition, while the training of ML models traditionally favors a centralized approach, scalability and privacy issues seem to lead towards a distributed one. The latter poses challenges to ML algorithms and the efficacy of learning itself.

Reproducing ML models poses several challenges arising from the intrinsic variability of the models themselves and the environment where they are trained. This problem is aggravated by their lack of standardization and common terminology.

The main goal of this work is to conceptualize and prototype a framework to train, evaluate and describe ML models, in a decentralized way, over immunogenetics datasets. This framework will promote model reproducibility and comparability, as well as its adaptability.

This work will start by implementing a federated/decentralized training framework over existing ML pipelines. Then, it will be possible to list and select potential dataset sources, aiming to provide an easy path to model adaptation and optimization.

Keywords: federated learning, decentralization, machine learning, immunology, immunotherapy, genetics

Resumo

Os contínuos avanços e crescente interesse em IA (Inteligência Artificial) no campo da saúde levantaram novas questões, nomeadamente a explicabilidade e a reprodutibilidade de modelos de ML (Machine Learning).

Adicionalmente, enquanto o treino de modelos de ML favorece tradicionalmente uma abordagem centralizada, questões de escalabilidade e privacidade tendem a levar a uma abordagem distribuída. Esta última apresenta desafios aos algoritmos de ML e à eficácia do treino em si.

A reprodução de modelos de ML apresenta vários desafios decorrentes da variabilidade intrínseca dos próprios modelos e do ambiente onde são treinados. Este problema é agravado pela falta de padronização e terminologia comum.

O principal objetivo deste trabalho é conceptualizar e prototipar uma framework para treinar, avaliar e descrever modelos de ML, de forma descentralizada, sobre conjuntos de dados imunogenéticos. Essa framework promoverá a reproducibilidade e comparabilidade dos modelos, bem como a sua adaptabilidade.

Este trabalho começará com a implementação de uma framework de treino federado/descentralizado sobre pipelines de ML existentes. De seguida, será possível listar e selecionar potenciais fontes de dados, esperando facilitar a adaptação e otimização dos modelos.

Palavras-chave: federated learning, decentralization, machine learning, immunology, immunotherapy, genetics

Acknowledgement

To my supervisor, Doutor Luiz Faria, for his support, guidance, input, and patience.

To Professor Artur Rocha and Alexandre Costa, for their thorough support, and management input.

To the immuneML team, who helped understand the bigger picture and consequences of this work, as well as their guidance throughout the project.

And finally, to my family and friends, who continuously showed their support through these roller-coaster times, and never failed to demonstrate their love.

Contents

List of Figures

List of Tables

List of Source Code

Chapter 1

Introduction

1.1 Introduction

The international iReceptor Plus consortium promotes human immunological data storage, integration, and controlled sharing for clinical and scientific work. Co-funded by the EU and the Canadian government, the project develops an innovative platform to integrate distributed repositories of Adaptive Immune Receptor Repertoire sequencing (AIRR-seq) data. This platform will improve personalized medicine and immunotherapy in cancer, inflammatory and autoimmune diseases, allergies, and infectious diseases (*Promote Integration of Large AIRR-seq Data* 2022).

This (AIRR-seq) data must adhere to the AIRR Community standards and protocols for generating, analyzing, depositing, exploring, and sharing such data. iReceptor Plus will be designed as a federated repository (AIRR Data Commons) network. Such a network facilitates data queries and advanced analyses through easy-to-use web portals (*Platform to Integrate Distributed Repositories of AIRR-seq Data* 2022).

1.2 Problem

While the training of ML models traditionally favors a centralized approach, scalability, bandwidth, and privacy issues reveal the distributed approach to be the most appropriate solution. This decentralized approach challenges ML algorithms and their learning efficacy. Still, it enables models to be created without a central data repository, having no data leave its original repository, thus maintaining compliance to local, specific privacy constraints (Gad 2020).

immuneML, described in section 2.3, is a data analysis framework developed by iReceptorPlus partners. This package already performs data encoding, which is considered a more (albeit not fully) private approach for the sharing of data in ML studies. This means actual data is never shared, but the meaningful information, needed for the ML to effectively work, is still shared, along with some metadata, connecting each data entry to at least the training flags, e.g. a certain disease being present or not.

A prototype has already been developed that reuses a selection of packages and modules from immuneML (section 2.3) to create a platform that performs a decentralized data encoding and workflow execution. This work is limited due to its inability to share trained models instead of encoded data, which, albeit encoding the data before sharing, still allows some information to reach the analyzing party. The whole interface and interaction with the application are also primitive and should be improved to allow for more reproducible results.

1.3 Motivation

New technology can spread quickly, sometimes even quicker than a third party's knowledge of making the technology reliable, reproducible or usable. Many challenges currently weigh down on the effective sharing of AIRR-seq data. First, the ample datasets storage and transfer, which can incorporate hundreds of millions of sequences (and hundreds of gigabytes) for a study, are time and resource-heavy. Also, journals or funding agencies do not consistently require data upload to public archives. Finally, the steps necessary for properly using the data by third parties should be standardized (Breden et al. 2017).

The processing AIRR-seq pipeline between the experiment and the data analysis is long and specialized. New metadata standards, specific to AIRR-seq methods, are required to effectively share and reanalyze this pipeline and data (Breden et al. 2017).

Chapter 2

State of the art

2.1 Background

In this section, some detail will be introduced about AIRR sequencing, which is the basis of the work and platforms developed, the iReceptorPlus project, within which this work was developed, and some information about the state of artificial intelligence in the immunotherapy field.

2.1.1 AIRR-seq

The many pathogens a person will contact during her life create a reaction from the adaptive immune system. This reaction is what protects people against diseases. B and T cells are the pillars of the adaptive immune system, and both show variable, antigen-specific receptors called B cell receptors (BCRs) and T cell receptors (TCRs), respectively. These receptors are generated by somatic gene-segment rearrangement, which produces unique, antigen-specific regions. An individual's combination of BCRs and TCRs is called the adaptive immune receptor repertoire (AIRR), responsible for recognizing various pathogens, auto-antigens, allergens, toxins, and tumors (Breden et al. 2017; Rubelt et al. 2017). These receptors and their sequences have been studied for decades, providing insights into the interactions between the immune receptors and antigens and helping with antibody engineering. Since HTS (High Throughput Sequencing) started being applied to AIIR sequences, experimental and computation techniques have advanced rapidly. These techniques provide information on how different factors, like age, genetic background, health status, and antigen exposure, variate the AIRR (Breden et al. 2017). AIRR-seq data is increasingly important in biomedical research and health care applications such as vaccine research, cancer immunotherapy, antibody discovery, Minimal-Residual Disease (MRD) detection, therapy response monitoring (Breden et al. 2017; Rubelt et al. 2017).

The massive amount of datasets and data analysis tools created a need for platforms to enable their management. Sharing and managing solutions started being presented by the AIRR community, like the iReceptor project (Rubelt et al. 2017). In recent years, the iReceptor Plus project set out to improve and work on its predecessor.

2.1.2 iReceptorPlus project

The iReceptor Plus project has a four-year duration and is co-funded by the EU and Canadian governments. The project comprises twenty partners from Europe, Canada, and the USA, including INESC TEC from Portugal. This platform aims to improve personalized medicine and immunotherapy by enabling researchers to store, share, and analyze massive datasets located in multiple storage facilities (*Platform to Integrate Distributed Repositories of AIRR-seq Data* 2022).

Usually, AIRR-seq datasets are stored and curated in separate and individual labs. The platform will ease the access and analysis of large amounts of AIRR-seq data, facilitating worldwide studies. This ability will promote the discovery of biomedical interventions that manipulate the adaptive immune system, aiding the personalization in medicine and immunotherapy in cancer, inflammatory and autoimmune diseases, infectious diseases and allergies (*Platform to Integrate Distributed Repositories of AIRR-seq Data* 2022).

AIRR-seq datasets can be colossal and very complex, so platforms like iReceptor and iReceptor Plus appeared to improve their usability. The iReceptor Plus framework will follow the AIRR Community guidelines and standards to increase interoperability and reproducibility within the area. The project is also focused on preserving the privacy of any and all sensitive data, intending to create a platform that allows for data analysis with minimal to no privacy concern. (*Platform to Integrate Distributed Repositories of AIRR-seq Data* 2022).

2.1.3 Artificial Intelligence and Immunotherapy

Machine Learning is able to import and process enormous datasets and generate models that fit not only the data they are trained on, but also other similar data inputs. Immunotherapy, on a molecular level, is assisted by ML in several fields (Jabbari and Rezaei 2019):

- **Classification** - Phenotype detection and classification are able to determine the presence and potential outcome of a disease. For instance, as some viral phenotypes are directly linked to drug resistance, the HIV-1 reaction and response to certain treatments can be accurately predicted. The classification of virus strains based on their phenotype features that are linked to resistance is also possible and made more efficient;
- **Image-based detection** - Phenotype detection, at both molecular and cellular level, can require less time and become less prone to a subjective judgment, when allied by ML-powered image-based detection systems. Human visual inspections are slow and unable to handle the enormous amounts of data some studies produce. ML methods can not only handle much higher orders of dataset sizes, but are also more unlikely to miss phenotype changes that a human;
- **Predictive model** - Predicting the probability of the MHC molecules presenting a certain peptide, which represent the ability of the immune system to bind with and recognize foreign substances, has been tested with ML models with very high accuracy. These predictions are some of the strongest AI tools in developing new therapeutic/prophylactic agents like vaccines.

There are also other fields within immunotherapy that can be aided by ML methodologies (Jabbari and Rezaei 2019):

- **Video-based detection** - Video-based algorithms are used to recognize the patient's face and mouth, as well as the medication being administered, aiming to determine whether the drug has been swallowed;
- **Trial eligibility** - AI's pattern-finding nature can help researchers find eligible patients by scanning their (digital) medical records. With this information, patient's data can

be compared and fitted against the study-specific criteria for eligibility and return the ideal subjects;

- **Simulate/Replace humans in trials** - When the right data is trained on an ML model, there is the possibility of AI replacing human patients on clinical trials. With many aspects of AI and ML under a gray ethical area, this replacement is considered very useful in the early stages of clinical trials.

Between all uses for ML methods in immunotherapy, AIRR-seq takes advantage of its classification and predictive properties. AIRR-seq ML pipelines are built to classify and predict on AIRR data, like the presence of a disease, exposure to pathogens or vaccines (*Mining adaptive immune receptor repertoires for biological and clinical information using machine learning - ScienceDirect* 2022).

2.2 Reproducibility

Machine learning in the life sciences has shown tremendous growth in the past years, primarily due to its ability to handle the scale and complexity of biological data. However, machine learning models are opaquer and more prone to learning bias than simpler models (Heil et al. 2021). With the increasing usage of ML for decision-making and knowledge generation, it becomes crucial to reproduce ML experiments. To trust and build upon any results, these must be reproducible. As with many science disciplines, ML is facing a "reproducibility crisis" (Hutson 2018).

Meanwhile, study (Gundersen, Shamsaliei, and Isdahl 2022) suggests that the leading commercial and academic machine learning platforms do not enable full reproduction potential. These platforms need extra work from researchers to make their work reproducible. Although reproducibility and replication discussions are mainly focused on traditional statistical models and results from randomized clinical trials, the same ideals should be implemented in machine learning studies (Beam, Manrai, and Ghassemi 2020).

Managing and storing each model checkpoint is essential to reproducibility because artifacts make it easy for ML team members to replicate and validate models. Data sources are also a crucial aid in helping other researchers fully reproduce ML experiments (Beam, Manrai, and Ghassemi 2020; Onose 2021).

Samuel (Samuel, Löffler, and König-Ries 2020) argues that the description and sharing of the research process and related data should follow FAIR data principles. The **FAIR** data principles are aimed to improve **F**indability, **A**ccessibility, **I**nteroperability, and **R**euse. These principles were created to improve the capacity of computational systems to find, access, interoperate, and reuse data with as little human interaction as possible (Wilkinson et al. 2016). To promote interoperability, shared ML research should be based on common terminology, and Samuel (Samuel, Löffler, and König-Ries 2020) presents a set of questions to help the standardization processs:

- Which hyperparameters were used in one run of the model?
- Which libraries and their versions are used in validating the model?
- What is the execution environment of the ML pipeline?
- How many training runs were performed in the ML pipeline?
- What is the allocation of samples for training, testing, and validating the model?

- What are the defined error bars?
- Which are the measures used for evaluating the model?
- Which are the predictions made by the model?

In the same paper (Samuel, Löffler, and König-Ries 2020), the author presents several challenges faced by scientists when trying to reproduce other work results:

- Unavailability, incomplete, outdated, or missing parts of source code;
- Unavailability of datasets used for training and evaluation;
- Unavailability of a reference implementation;
- Missing or insufficient description of hyperparameters that need to be set or tuned to obtain the exact results;
- Missing information on the selection of the training, test, and evaluation data;
- Missing information of the required packages and their version;
- Tweaks performed in the code not mentioned in the paper;
- Missing information in methods and the techniques used, e.g., batch norm or regularization techniques;
- Lack of documentation of preprocessing steps, including data preparation and cleaning;
- Difficulty in reproducing training of large neural networks due to hardware requirements.

In another study (McDermott et al. 2019), reproducibility is divided into three replicability ranks, depending on their application in different environments:

- **Technical Replicability** – where a result can be fully replicated and match the results reported by the original researcher;
- **Statistical Replicability** – where re-sampled conditions (e.g., change in train/test splits) yield different configurations but do not statistically affect the original results;
- **Conceptual Replicability** – where the results maintain statistical relevance in the same conceptual field as the purported effect. This reproducibility requirement is harder to meet as the conceptual horizon of generalizability increases.

Similarly, but having in mind the easiness of reproducibility, study (Heil et al. 2021) creates the following three standards:

- **Bronze** – where data, models, and source code are published and downloadable;
- **Silver** – where Bronze is met, dependencies are set up in a single command, key details from the analysis are recorded, and the analysis components are set to deterministic;
- **Gold** – where Silver is met, and the entire analysis is reproducible with a single command.

As shown in Figure 2.1, ML4H lags behing all other ML fields in all reproducibility evaluation metrics except on the statistical replicability front, where ML4H reports the inclusion of proper statistical variance. The biggest differences are:

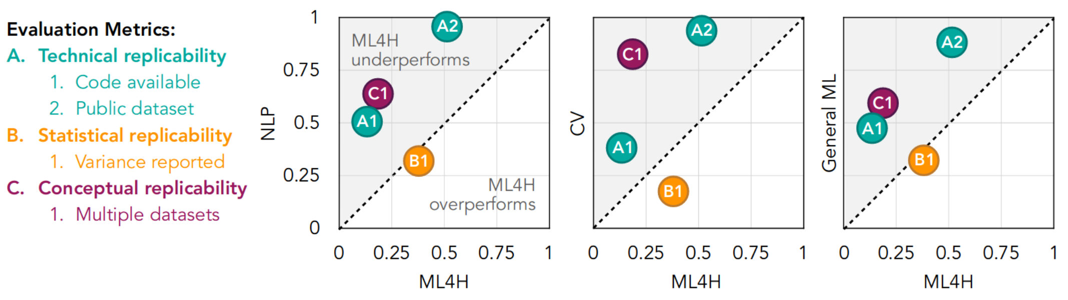

Figure 2.1: Comparison of Technical, Statistical, and Conceptual replicability between ML4H and other ML fields (McDermott et al. 2019)

	NLP	CV	genML	ML4H
Usage of public datasets	>90%	>90%	77%	51%
Public code release	50%	37%	—	13%
Result variance description	33%	17%	32%	38%
Multiple dataset testing	66%	83%	—	19%

Table 2.1: Comparison of replicability guidelines between ML4H and other ML fields (McDermott et al. 2019)

With these questions, problems, and goals in mind, studies (Banna et al. 2021; Pineau et al. 2020) share some checklists to go over before the release of an ML work, where the potential problems and considerations for the model re-implementation should arise. Study (Pineau et al. 2020) presents a shorter list, as seen in Figure 1, in Appendix, whereas study (Banna et al. 2021) divides these checks in three main blocks that follow:

- General Checks:
  - Model Purpose;
  - Code Availability;
  - Language/Framework/Libraries Used;
  - Networks Referenced.
- Model and Design Checks:
  - Model Architecture;
  - Model Sub-Networks;
  - Model Building Blocks;
  - Custom Layers;
  - Loss Function;
  - Output Structure.
- Training and Evaluation Checks:
  - Dataset Used;
  - Pre-Processing Functions;

  - Output Processing Functions;
  - Testing and Target Metrics;
  - Training Steps.

Both lists show slightly different yet complementary approaches for a model reproducibility check. When used together, there will be a high degree of confidence in whether the model is reproducible.

2.3 immuneML

immuneML is an open-source platform for AIRR ML (Figure 2.2). immuneML enables AIRR-seq data ML study, from preprocessing to model training and interpretation. This platform has four main scopes, all of which are used to perform data analysis, with or without the usage of ML (*immuneML documentation* 2022; Pavlović et al. 2021):

- **Training ML models** – allows for repertoire or receptor sequence classification, like disease or antigen binding prediction, respectively. Different ML settings, like data preprocessing steps, encodings, and ML models and their hyperparameters, can be compared using nested cross-validation for performance measurement;
- **Exploratory analysis of datasets** – allows for the application of preprocessing, encoding, and plotting descriptive statistics without training ML models;
- **Simulating** – allows for the simulation of immune events, like disease states, and subsequent insertion into a dataset. ML settings can then be compared with known immune signals implanted in a dataset, creating a ground truth benchmarking dataset;
- **Applying trained ML models** – allows for the application of already trained ML models to new datasets, even when class labels are not known.

To start an analysis, immuneML requires a specification file, namely a YAML file. Looking at (Figure 2.2), this file defines the analysis settings: **Dataset**, **Preprocessing**, **Encoding**, **ML method**, **Report** or **Simulation**. The file also defines an instruction: **TrainMLModel**, **ExploratoryAnalysis**, **Simulation**, **MLApplication**, **DatasetExport** or **SubSampling**. A result HTML file is generated which may include **models**, **predictions**, **plots** or **datasets**. immuneML can be used via the command line or a web interface through Galaxy (Pavlović et al. 2021).

2.3.1 Usage

This section walks through the main flows to use immuneML, as shown on (*immuneML documentation* 2022).

Analysis specification

In immuneML, an analysis is specified through a fixed structure YAML file. The file defines the execution of different tasks, such as training ML models, data simulation or exploratory analysis. This specification consists of three parts (*immuneML documentation* 2022):

- **Definitions** – are the components that will be used inside the Instructions. They can be datasets, preprocessing steps, encodings, ML methods or reports. The user

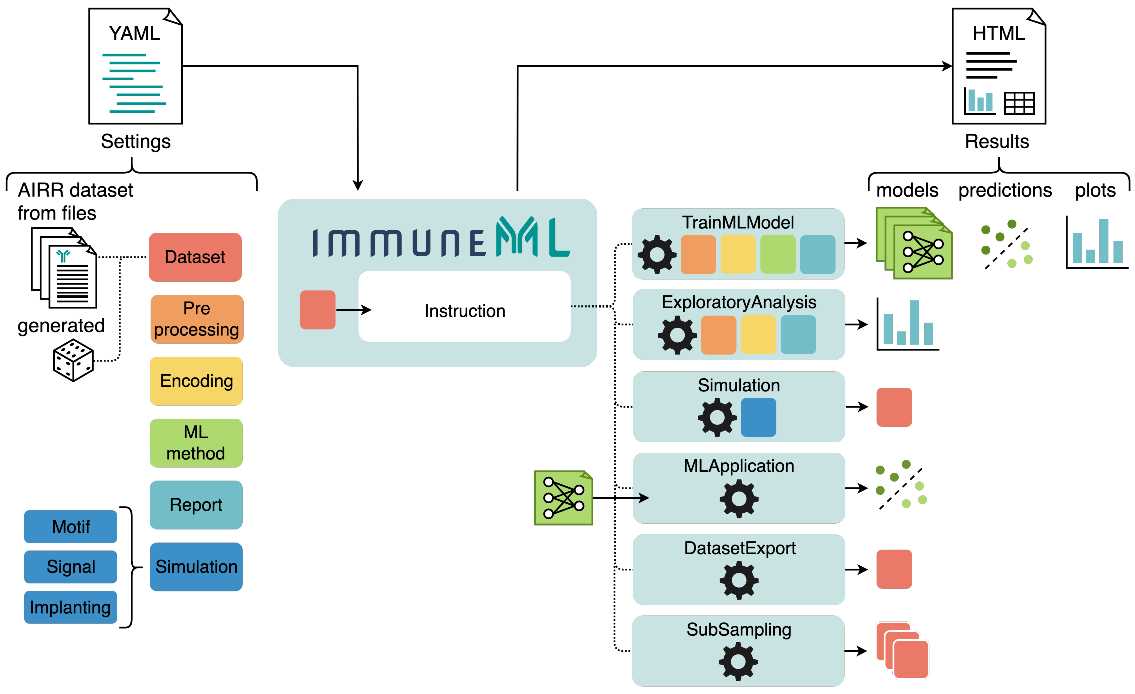

Figure 2.2: Overview of how immuneML analyses are specified (Pavlović et al. 2021)

must identify each component using any desired, unique key. This unique key is later referenced in the instructions;

- **Instructions** – are components that describe the type of analysis to be made and all its respective and specific settings. If an output from an instruction is needed for another instruction, two separate immuneML flow runs need to be made;
- **Output** – currently only HTML is supported.

Data import

To start an analysis in immuneML a dataset must be imported. immuneML supports three types of datasets (*immuneML documentation* 2022):

- **Repertoire datasets** – used on per repertoire predictions, like disease state prediction;
- **Sequence datasets** – used on single immune receptor chain prediction, like antigen specificity;
- **Receptor datasets** – used on paired receptor chain prediction. Is the paired version of Sequence datasets.

A vast array of formats are allowed for import, including **AIRR** (Christley et al. 2020; Vander Heiden et al. 2018), **MiXCR** (Bolotin et al. 2015; Rosenfeld et al. 2018), **VDJdb** (Bagaev et al. 2020; Goncharov et al. 2022; Shugay et al. 2018), among others, with non-supported datasets able to be imported with a **Generic** option. Some metadata is required such as the filename and subject_id. When importing data, some fields are mandatory like the dataset key name, format and path (*immuneML documentation* 2022).

Data generation

In order to try out some functionalities, or create baseline datasets for ML model comparison, immuneML offers the possibility of generating random datasets of immune receptor sequences, immune receptors or immune repertoires. This option comes as an alternative to loading an existing dataset in an analysis specification. Dataset size, sequence lengths and labels can be specified by the user (*immuneML documentation* 2022).

Focusing on the Repertoire datasets, the datasets field in the YAML file Definitions should be designed as shown in Listing 2.1.

```yaml
datasets:
  my_dataset:
    format: RandomRepertoireDataset
    params:
      repertoire_count: 100 # number of repertoires to be generated
      sequence_count_probabilities: # the probabilities have to sum to 1
        100: 0.5 # the probability that any repertoire will have 100 sequences
        120: 0.5 # the probability that any repertoire will have 120 sequences
      sequence_length_probabilities: # the probabilities have to sum to 1
        12: 0.33 # the probability that any sequence will contain 12 amino acids
        14: 0.33 # the probability that any sequence will contain 14 amino acids
        15: 0.33 # the probability that any sequence will contain 15 amino acids
      labels: # metadata that can be used as labels, can also be empty
        HLA: # label name, any name can be chosen (the probabilities per label value have to sum to 1)
          A: 0.6 # the probability that any generated repertoire will have HLA A
          B: 0.4 # the probability that any generated repertoire will have HLA B
```

Listing 2.1: Dataset generation YAML definition (*immuneML documentation* 2022)

The defined dataset can be exported with the Listing 2.2 instruction set.

```yaml
instructions:
  my_dataset_export_instruction:
    type: DatasetExport
    datasets: [my_dataset] # list of datasets to export
    export_formats: [AIRR, ImmuneML] # list of formats to export the datasets to.
```

Listing 2.2: Dataset generation YAML instructions (*immuneML documentation* 2022)

Train and assess

Receptor or repertoire classification ML models can be trained and assessed with immuneML. The settings for hyperparameters influence the training process and can not be optimized by training. Since preprocessing steps and encoding also affect the model's performance, they can be considered hyperparameters during training. Several methods, like nested cross-validation, and fixed splits, are supported, which allow for hyperparameter optimization and an overall unbiased assessment (*immuneML documentation* 2022). The ML workflow is depicted in Figure 2.3.

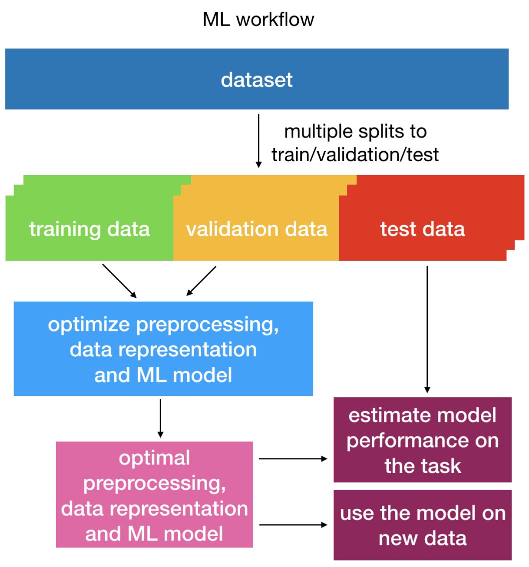
ML workflow
dataset
multiple splits to train/validation/test
training data
validation data
test data
optimize preprocessing, data representation and ML model
optimal preprocessing, data representation and ML model
estimate model performance on the task
use the model on new data

Figure 2.3: Overview of the training process of an ML classifier: hyperparameter optimization is done on training and validation data and the model performance is assessed on test data (*immuneML documentation* 2022)

Apply trained ML model

The optimal ML settings retrieved from training (trained model, encoding and/or preprocessing steps) are exported for each label. New analysis are able to predict labels on new datasets by using these settings. This means the model will only work if the new dataset is preprocessed and encoded in the same way. If other major differences are present in dataset generation, like different sequencing platform, the model performance may deteorate (*immuneML documentation* 2022).

Exploratory analysis

Preprocessing steps, encodings and reports can be explored without running an ML algorithm. Each analysis must contain a dataset and a report and multiple analysis can be specificated in the same instruction (*immuneML documentation* 2022).

The definition of a default analysis based on an imported dataset, an (arbitrary) encoder, and two different (also arbitrary) reports is shown in the Listing 2.3 YAML configuration.

```yaml
definitions:
  datasets:
    # imported datasets
    my_dataset: # user-defined dataset name
      format: AIRR
      params:
        metadata_file: path/to/metadata.csv
        path: path/to/data/

  encodings:
    my_regex_matches:
      MatchedRegex:
        motif_filepath: path/to/regex_file.tsv

  reports:
    my_seq_lengths: SequenceLengthDistribution # reports without
   parameters
    my_matches: Matches
```

Listing 2.3: Exploratory analysis YAML definition (*immuneML documentation* 2022)

The encoder definition is present to allow for two different analysis to be performed on the same dataset, as shown in the Listing 2.4 YAML snippet.

```yaml
instructions:
  my_instruction: # user-defined instruction name
    type: ExploratoryAnalysis
    analyses:
      my_analysis_1: # user-defined analysis name
        dataset: my_dataset
        report: my_seq_lengths
      my_analysis_2:
        dataset: my_dataset
        encoding: my_regex_matches
        report: my_matches
```

Listing 2.4: Exploratory analysis YAML instructions (*immuneML documentation* 2022)

Various analysis can be run on the same dataset if correctly encapsulated in their respective analysis object.

Signal simulation and motif implantation

Immune events on repertoires can have their effects simulated by implanting signals in the original dataset. The original dataset can be any repertoire dataset, whether experimental or simulated (*immuneML documentation* 2022).

Immune events within a simulation are represented by three classes, as seen in Figure 2.4:

- **Motif** - the sequence of amino acids to be implanted in an AIRR repertoire dataset. This is described by a seed and instructions on how that seed is instantiated, working in a similar way to regular expression algorithms but with focus on known genetic properties;

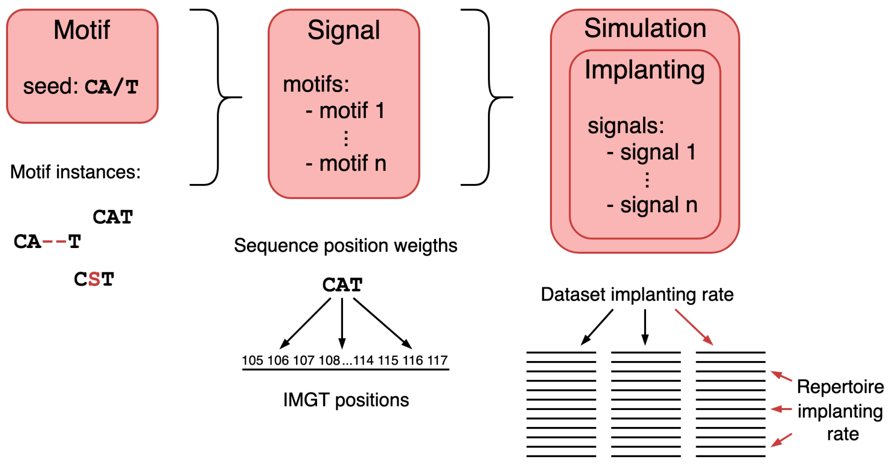

Figure 2.4: How different concepts in a simulation relate: Motif, Signal, Implanting, and Simulation (*immuneML documentation* 2022)

- **Signal** - the list of different motifs aligned with the sequence position weights, which represent the probability of a motif to be implanted in the given sequence positions;
- **Implanting** - the interface connecting the signal to the actual dataset simulation. Here the dataset and repertoire implanting rates, for each signal (or combination of signals), are defined. Representing the percentage of repertoires and receptor sequences which will contain the signal, the dataset and repertoire implanting rates, respectively, can be varied to understand how a signal presence can affect a study outcome.

In the Listing 2.5 is the documentation guide on building a simulation YAML specification file, with instructions being very similar to previous examples (*immuneML documentation* 2022). Initially, a simple motif is created with the seed *AAA*, meaning that string will be implanted in the sequence. A signal is created pointing to the previously created motif, as well as the possible implanting positions and their relative weight, meaning the motif can be implanted in the positions *109*, *110*, *111*, or *112*, with probabilities of *0.1*, *0.2*, *0.5*, and *0.1*, respectively. The simulation part of the file defines the signal that is being implanted and its details. The details include the dataset implanting rate, that defines the percentage of repertoires that will have the signal implanted, and the repertoire implanting rate, that defines the percentage of sequences, inside each repertoire, implanted with the signal.

```yaml
definitions:
  motifs:
    my_simple_motif: # a simple motif without gaps or hamming distance
      seed: AAA
      instantiation: GappedKmer

  signals:
    my_signal:
      motifs:
        - my_simple_motif
      implanting: HealthySequence
      sequence_position_weights:
        109: 0.1
        110: 0.2
        111: 0.5
        112: 0.1

  simulations:
    my_simulation:
      my_implanting:
        signals:
          - my_signal
        dataset_implanting_rate: 0.5
        repertoire_implanting_rate: 0.25
```

Listing 2.5: Simulation YAML (*immuneML documentation* 2022)

2.4 Federated Learning

Federated Learning (FL) has its origin in the Google Keyboard in 2016, where several Android devices collaboratively learn. The potential to apply this technology to other edge devices brings the possibility of revolution in domains such as healthcare (Mammen 2021).

Traditional, centralized ML algorithms require the data to be aggregated in a central repository for model training. Regulatory restrictions on sensitive data sharing are the main challenge for genetic information-based healthcare applications. FL eliminates the need for raw data sharing, allowing ML model training with data distributed between multiple data storage facilities. After a global model is built and shared across nodes, each node can train the model based on its data, and the updates to the model are later joined in an aggregation server. This process is repeated until a global model convergence criterion is met (Choudhury et al. 2020). The process through which the global model is generated from the local models is called Federated Averaging Algorithm (Gad 2020).

FL has shown merit in various real-world applications, like image classification and language modeling. In healthcare applications, where data is personal, sensitive, and highly regulated, FL shows particular relevance (Choudhury et al. 2020).

2.4.1 Types of Federated Learning

Federated Learning can be divided into the following types (Mammen 2021; Yang et al. 2019):

- **Vertical Federated Learning** - used when each device contains a dataset with a different set of features but the same group of sample instances;

- **Horizontal Federated Learning** - opposing to Vertical Federated Learning, is used when each device contains a dataset with the same set of features but for a different group of sample instances;
- **Federated Transfer Learning** - used for adding a new feature to a model that was already fitted to previous features.

2.4.2 Workflow

There are four main steps to the federated learning workflow cycle, all of which are repeated, in order, for as many times as the user or convergence mechanism determine (Bonawitz et al. 2019; Li, Sahu, Talwalkar, et al. 2020; Mammen 2021; Rieke et al. 2020; Rodríguez-Barroso et al. 2020):

- **Client selection** - Server pings and selects the desired amount of clients from those connected, using a (possibly random) algorithm. (Khan et al. 2020; H. Yu et al. 2020; T. Yu et al. 2020) study in more depth the intricacies of the client selection process;
- **Parameter broadcast** - Server sends the parameters of the global model to the picked clients;
- **Local model training** - The clients download the global model parameters and start a new round of model training using local data, sending the results back to the server;
- **Model aggregation** - The server receives the new models with the updates and aggregates them together to generate a new global model.

2.4.3 Usual applications

There are four main areas where federated learning has the most impact, according to (Mammen 2021):

- **Healthcare** - The biggest source of healthcare data for ML model training are the Electronic Health Records. Training data quality is essential for model performance and when these models are created using data from only one hospital, the predictions tend to have a higher probability of being biased. To try to generalize models, data needs to be shared across institutions which, with healthcare data being sensitive and restricted, sharing datasets may not be an option and federated learning can be an important tool in allowing for greater, privacy-respecting research;
- **Transportation** - Technology and sensor integration in vehicles has allowed for an ever bigger amount of data being collected and available for ML studies. Vehicle and traffic management and autonomous driving are some of the areas where ML models can have a big impact and, due to the centralization of the training process, vehicles can have a hard time adapting to the plethora of unique situations each of them faces every ride. Federated learning can not only protect the driving data of drivers, but also improve the labeling accuracy of the different features;
- **Finance** - The financial sector can also benefit from federated learning, in areas such as loan risk assessment. Sharing information between financial institutions and commerce companies can allow for very sensitive data to be exposed, and federated learning could be used to create a risk assessment ML model that factors in all that information;

- **Natural Language Processing** - NLP is one of the most common ML applications, with the downside of needing really massive datasets to train precise models. The data for this type of model can be found in every common edge device, from phones and tablets, to headphones and smartwatches. Again, the data is highly available but highly confidential, so federated learning comes again as a strong candidate to tackle this matters. A working example of this implementation strategy is shown in (Bernal n.d.).

2.4.4 Strategies

For a Federated Learning pipeline to work, it must select a way in which it will merge

FedAvg

FedAvg tries to train a shared model across devices, by attempting to minimize a global loss. This overall loss is calculated by a weighted average of all client node's losses (McMahan et al. 2017).

$$f(w) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w)$$

This function considers K clients, each one running its own loss function, $F_k(w)$. It also shows the weight of each client's loss is calculated by dividing the client's dataset size, n_k, by the sum of all client's dataset sizes, n. This approach means that devices with bigger datasets at training time will get bigger weights and, therefore, more importance in the shared model aggregation (McMahan et al. 2017).

Although this strategy works well in practise, and does well in mantaining the model importance based on the amount of data provided, it is not the best solution for every scenario. Federated Averaging makes these simplifying assumptions that may hinder the learning process in specific situations:

- It assumes every client will be able to perform a predetermined number of training epochs in about the same time. In a situation with a big performance spread, the convergence speed and overall training performance can be hindered by slower devices. There is also the possibility that most devices connected are slow and their data is needed, so there is not the possibility of just dropping them for faster clients;
- Convergence is not guaranteed on very heterogeneous data, as the weighted average favors the clients with the biggest datasets, making the models biased towards these devices' data.

For comparison in the next strategies, the following function will be used, replacing the client weight fraction with p_k, representing that same proportion.

$$\min_{w} f(w) = \sum_{k=1}^{m} p_k F_k(w)$$

FedProx

FedProx, with *Prox* meaning proximal, tries to tackle the discrepancy in device performance by allowing devices to do variable work, meaning it will approximate the devices performances. While it seems that this strategy will allow for faster devices to perform more training epochs and, therefore, change their weights far more than a slower counterpart, FedProx itriesntroduces a regularization term, or proximal term, to its loss function (Li, Sahu, Zaheer, et al. 2020).

$$\min_{w} h_k(w; w^t) = F_k(w) + \frac{\mu}{2}||w - w^t||^2$$

Having the current round t, and the hyperparameter μ, the regularization term, $\frac{\mu}{2}||w - w^t||^2$ penalizes large changes in model weights more. This can prove helpful in the convergence of highly heterogeneous data, as this proximal term can be seen as a mechanism to stop the model from changing too much in any given device. The hyperparameter μ can be tuned in order to change the penalty magnitude and sensitivity.

qFedAvg

qFedAvg intends to bring fairness to the shared model learned in a federated learning pipeline, meaning it tries to make the model perform in the same way on all devices (Li, Sanjabi, et al. 2020).

$$\min_{w} f_q(w) = \sum_{k=1}^{m} \frac{p_k}{q+1} F_k^{q+1}(w)$$

By using the same data proportion as FedAvg, p_k, this strategy penalizes the worse performing devices more, in an attempt to make the model improve its performance on those devices. By tuning the hyperparameter q, it becomes possible to affect how much the worse devices dominate the overall model. Increasing q will make these devices dominate the model more, making the model more fair, and vice-versa.

perFedAvg

perFedAvg tries to create a personalized model for each client with few local epochs (Fallah, Mokhtari, and Ozdaglar n.d.).

$$\min_{w} f(w) = \sum_{k=1}^{m} p_k F_k(w - \alpha \nabla F_k(w))$$

This method differs from FedAvg by not calculating the client's loss on its current weights but on the weights after a gradient descent step, as seen by the replacement of $F_k(w)$ with $F_k(w - \alpha \nabla F_k(w))$

2.4.5 Frameworks

There already exist some frameworks that enable Federated Learning from existing ML pipelines, and a few will be explored in this section.

Flower

Flower is a Federated Learning framework that focuses on the execution of large-scale experiments, simulate and deploy to highly heterogeneous devices, for system oriented research within the domain. Flower is a ML framework and language-agnostic platform, meaning that with some degree of tweaking, most ML pipelines can be federated and decentralized (Beutel et al. 2022).

TensorFlow Federated

TensorFlow Federated is a computational framework for decentralized data, which includes ML algorithms. This framework works on top of Google's TensorFlow, and attempts to enable Federated Learning practices to it. This pipeline has been used to train language models from devices' keyboards, without ever releasing or uploading their (private) data (*TensorFlow Federated* 2022).

PySyft

PySyft is an open source library that intends to provide more security and privacy to Data Science operations in Python. Developed by OpenMined, this framework can enable Federated Learning studies in pipelines built over major ML libraries, such as Tensorflow, and PyTorch (Ryffel et al. 2018).

Comparison

Although all platforms allow for both simulating the FL pipeline and running the pipeline in actual devices, when it comes to scalability all frameworks lose to Flower. The fact that this platform is ML framework-agnostic gives it a serious advantage to all competitors, as you can use the same implementation, and very similar code, to run different studies, requiring different ML tools, in the same devices (and perhaps same data).

Given Flower's Communication and Language-agnostic nature, it allows for its client to be ran on any type of device (e.g., phone, laptop, server, embedded), while TensorFlow Federated and PySyft expect the framework to provide the runtime environment to the client.

Chapter 3

Solution

3.1 Proposed solution

As mentioned in the problem definition (section 1.2), a primitive prototype platform was developed that allows for decentralized workflows based on immuneML. This prototype is divided into three main components:

- **Data Provider Node** – contains sensitive local data. When its data is requested by the **Analysis Node**, it encodes the data in an irreversible way before sending it;
- **Analysis Node** – requests **Data Provider Nodes** and receives encoded data. Node where model training occurs. Before training, the data is merged, splited, normalized, and standardized;
- **Server** – **Data Provider Nodes** and **Analysis Nodes** are registered here. Facilitates connections between them.

Meetings were held with members of the immuneML development and management teams, and the clear path towards more accessible and private analysis using the platform was discussed. While the created prototype already works in a decentralized way, the model is trained in a single, central node, where all training data has to be placed before starting the study. This either sends relevant (and possibly sensitive) data to the analysis node, since that data must contain the information the model will be trained on, or will share very poor quality data, as to not leak any information from its origin, resulting in bad model quality.

The solution presented here consists of federating the existing prototype, meaning that a model training and evaluating implementation will be added to each DPN, and model aggregation and evaluation modules will be added to the AN. The Server will still provide an easy path for ANs to discover which DPNs are online. This decentralization process will allow for models to be trained on data from all the selected entities, without ever sharing actual data, thus contributing to a higher preservation of privacy of the analysis.

Some modules from immuneML that were reused in the prototype, as well as some new ones, will be refactored and used in the new federated framework. These packages will enable the importing, encoding, normalization, merging, and splitting of the data, as well as training, assessing and reporting on the ML model, in both DPN and AN pipelines. The Server will continue to allow for ANs and DPNs to discover each other, but will no longer take part in the connection process, with this becoming direct communication between them.

3.2 ImmuneML package analysis

Initially, the immuneML analysis flow was decomposed in order to understand how they are defined and how the different components interact with each other. This flow is defined in a YAML specification file, as mentioned in section 2.3. The immuneML website documentation section 2.3 sheds some light on the multiple dependencies that exist within this specification.

In order to better understand how immuneML components interact between them, they must first be well identified. Three main groups were identified as the highest level specification division: Definitions, **Instructions** and **Output**. Any analysis type can be defined as **Instructions**, while the actual running of the analysis is defined by the content of the **Definitions** block. The **Output** is limited to HTML so it will not be discussed in its own section.

3.2.1 Definitions

This section shows the components to be used in the remaining of the file and on the analysis, as well as their relations. These components are: **Datasets**, **Encodings**, **ML Methods**, **Preprocessing Sequences**, **Reports**, **Simulations**, **Signals**, and **Motifs**. In order to use some components, others are required or able to be set, according to the following dependencies:

- **Encodings** and **Preprocessing Sequences** – depends on **Datasets**;
- **ML Methods** and **Reports** – depend on **Encodings**;
- **Signals** – depends on **Motifs**;
- **Simulations** – depends on **Signals** and **Motifs**.

3.2.2 Instructions

This section conjugates the analysis steps with the components defined in **Definitions**, as well as the instructions parameters. The possible steps in this section are: **DatasetExport**, **ExploratoryAnalysis**, **MLApplication**, **Simulation**, **Subsampling**, and **TrainMLModel**. They correlate with the **Definitions** as follows:

- **DatasetExport** – depends on **Datasets**;
- **ExploratoryAnalysis** – depends on **Datasets**, **Encodings** and **Reports**;
- **MLApplication** – depends on **Datasets** and **Encodings**;
- **Simulation** – depends on **Datasets** and **Simulations**;
- **Subsampling** – depends on **Datasets**;
- **TrainMLModel** – depends on **Datasets**, **Encodings**, and **ML Methods**.

As the solution is based on ML model training, we can identify **Datasets**, **Encodings**, and **ML Methods** as vital in its development.

3.2.3 Analysis dependencies

As mentioned in section 2.3, **Datasets** can be categorized as **Repertoire Dataset**, **Receptor Dataset**, and **Sequence Dataset**. Figure 3.1 presents how the the analysis-pertinent modules are distributed. The dependencies between the **Encoders** and the **Dataset** categories, as well as between the **ML Methods** and the **Encoders**, are represented in Figure 3.1.

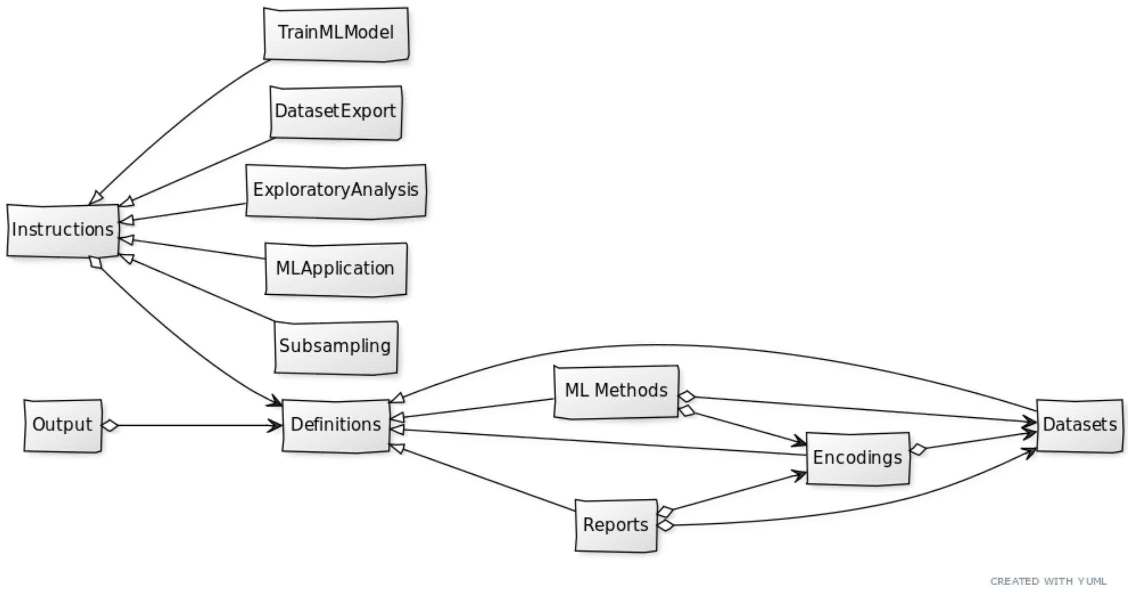

Figure 3.1: immuneML analysis specification diagram

Encoders

There are fourteen encoders in immuneML: AtchleyKmer, CompAIRRDistance, CompAIRRSequenceAbundance, DeepRC, Distance, EvennessProfile, KmerFrequence, MatchedReceptors, MatchedRegex, MatchedSequences, OneHot, SequenceAbundance, TCRdist, and Word2Vec. All encoders work on **Repertoire Datasets**, but only OneHot works for **Receptor** and **Sequence Datasets**. **EvennessProfile**, **KmerFrequence** and **OneHot**, **Word2Vec**, **KmerFrequency**, or **EvennessProfile** encoders are already implemented in the prototype, the solution aims to implement **OneHot** as to allow for a wider variety of Dataset category acceptance.

ML Methods

- **AtchleyKmerMILClassifier** – depends on **AtchleyKmer** encoder;
- **DeepRC** – depends on **DeepRC** encoder;
- **KNN** – depends on **OneHot**, **Word2Vec**, **KmerFrequency**, **EvennessProfile**, or **Distance** encoders;
- **LogisticRegression** – depends on **OneHot**, **Word2Vec**, **KmerFrequency**, or **EvennessProfile** encoders;
- **ProbabilisticBinaryClassifier** – depends on **SequenceAbundance** encoder;

- **RandomForestClassifier** – depends on **OneHot**, **Word2Vec**, **KmerFrequency**, or **EvennessProfile** encoders;
- **ReceptorCNN** – depends on **OneHot** encoder;
- **SVM** – depends on **OneHot**, **Word2Vec**, **KmerFrequency**, or **EvennessProfile** encoders;
- **TCRdistClassifier** – depends on **TCRdist** encoder.

KNN, **Logistic Regression**, **RandomForestClassifier**, and **SVM** methods are already implemented in the prototype. To match the **OneHot Encoder**, **RandomForestClassifier** will aid testing and the **KNN** and **ReceptorCNN** are aimed to be implemented.

3.2.4 Simulation dependencies

Datasets can be simulated, as mentioned in section 2.3. For a simulation pipeline to run on immuneML, the **Simulation**, **Signal**, and **Motif** fields must be defined. Their dependencies are shown in Figure 3.2.

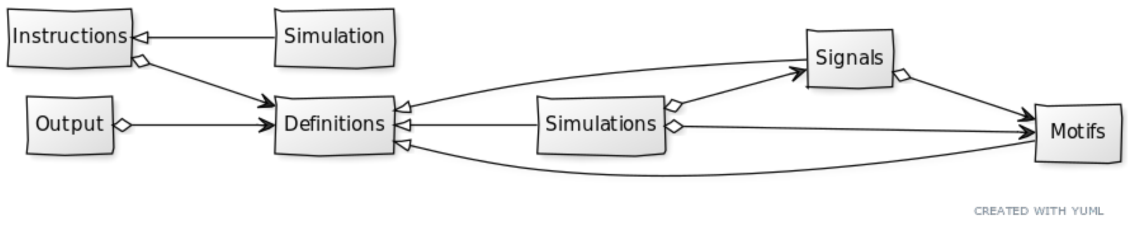

Figure 3.2: immuneML simulation specification diagram

3.3 ImmuneML reference pipeline

To develop a federated ML framework, a base pipeline from immuneML was selected as the base of this work. Initially, a summary of the packages and modules that will be used from immuneML are presented, then the pipeline that is going to be processed for decentralization is explored in greater detail.

3.3.1 Packages and methods used

This section will describe the modules from immuneML used in this project to implement the training, assessment, and reporting pipeline. These modules are used to make the federated framework compatible with the remaining workflows of immuneML. In Table 3.1 we explore where the different modules are located within immuneML.

- **NormalizationType** - Normalization of the train and test encoded datasets is selected and applied with the help of this module;
- **RepertoireDataset** - The EncodedData module can create a RepertoireDataset, needed for training, assessment, and reporting on an ML model;
- **Dataset** - The AIRR format is used throughout this project, and this module allows for the dataset to be stored after importing it;

Packager	Modules
analysis.data_manipulation	NormalizationType
data_model.dataset	RepertoireDataset
	Dataset
data_model.encoded_data	EncodedData
encodings	EncoderParams
encodings.kmer_frequency	KmerFrequencyEncoder
encodings.preprocessing	FeatureScaler
environment	Label
	LabelConfiguration
IO.dataset_import	AIRRImport
ml_methods	MLMethod
	LogisticRegression
ml_methods.util	Util
ml_metrics	Metric
reports.ml_reports	Coefficients
	CoefficientPlottingSetting
workflow.steps	MLMethodAssessmentParams
	MLMethodAssessment
	MLMethodTrainerParams
	MLMethodTrainer

Table 3.1: List of packages and respective modules used from the immuneML framework

- **EncodedData** - This module contains an object capable of storing the datasets after the encoding process, allowing it to be aligned with immuneML data standards;
- **EncoderParams** - To be able to use an Encoder, the EncoderParams object must be defined with the Encoder-specific parameters;
- **KmerFrequencyEncoder** - This module allows for data to be processed with the KmerFrequency Encoder;
- **FeatureScaler** - Used to normalize and scale data before training. For the normalization process, it takes the NormalizationType as input;
- **Label** - This module contains the Label object for the Encoders, where it usually consists of the variable to predict or classify on;
- **LabelConfiguration** - Encapsulation of Label objects for use with Encoders;
- **AIRRImport** - Used to load datasets from a data storage to a Dataset object;
- **MLMethod** - Abstract module where the common and useful ML method functions are defined. Actual used methods inherit this class, allowing for immuneML's easy method integration;
- **LogisticRegression** - Wrapper for sklearn's LogisticRegression ML method. This class iherits from MLMethod and applies method-specific tasks to it;
- **Util** - An utilitary class for various helpful functions;
- **Metric** - Object used to specify Metrics for training, assessment, and reporting;

- **Coefficients** - This module encompasses a reporting mechanism, that generates a barplot from the coefficients' values of the given model;
- **CoefficientPlottingSetting** - Configuration needed to plot a model with the Coefficients module;
- **MLMethodAssessmentParams** - Parameters needed for the assessment of an ML model;
- **MLMethodAssessment** - This class configures and runs the assessment of an ML model;
- **MLMethodTrainerParams** - Parameters needed for the training of an ML model;
- **MLMethodTrainer** - This module configures and runs the training of an ML model.

3.3.2 Pipeline specification

The immuneML centralized pipeline steps are the following:

- **1. Define encoding arguments** - Initially, the encoding parameters are set in a key-value dictionary. For *encoding* the option *KmerFrequency*, for *sequence_encoding* the value *CONTINUOUS_KMER*, for *reads* the value *UNIQUE*, for *sequence_type* the value *AMINO_ACID*, and for k a value of *3* was chosen. Essentially, the encoder groups each sequence into k-mers of size 3 (k), meaning it encapsulates all continuous sub-strings of the specified length;
- **2. Import dataset** - The dataset is imported from the data storage, by receiving a path to the metadata and repertoires' directories;
- **3. Encode dataset** - After importing, the system immediately runs the encoding steps with the above configuration. The datasets are then only used inside the system after encoding, which avoids any original data leak;
- **4. Add dataset** - The encoded datasets are added to an EncodedDatasets object, that can hold several datasets as needed by the analysis;
- **5. Merge datasets** - If more than one dataset is added to the EncodedDatasets object, they are merged into a single, bigger dataset in this step;
- **6. Define split, scale, and normalize parameters** - At this point, the parameters for data preprocessing are defined. The *normalization type* is set to *max*, the *scaling_to_zero_mean* flag is set to *False*, and the *split percentage* is set to *0.1*, which translates to ten percent of the dataset being used for testing and ninety percent for training;
- **9. Split, scale, and normalize data** - This step takes the previously defined parameters and processes the data prior to model training;
- **10. Initialize ML model** - This step defines the ML method that is being used along with its parameters, and how it should be initialized. For this pipeline the *Logistic Regression* method was chosen. This method will take various iterations to train the model, hence the parameter *max_iter* becoming important to variate this, and understand its effect on the final model;

- **11. Train ML model** - After initializing the model, it is used, along with the encoded data meant for training, and the training process starts;
- **12. Assess ML model** - After training a model, it can be measured against the part of the encoded data meant for testing, returning the accuracy and loss values of the model, when tested against that portion of data;
- **13. Report on ML model** - This step reads the trained model and builds a data structure based on the coefficient values of the *Logistic Regression* model. Given an output path, an HTML file is generated with a bar graph comparing the coefficient values of the model. This can show how a model was able to have a bigger coefficient for the injected k-mers section 3.4, in comparison to others.

3.4 Data gathering and identification

As explained before, ML model training depends on the encoding of datasets, which have a tendency to be very large. In order to obtain more manageable and predictable data, for faster testing, and data modeling, the immuneML dataset generation feature was used. As mentioned in section 2.3, using the YAML specification file, it is possible to generate a random dataset, and then implant immune signals inside it.

3.4.1 Test case definition

After some meetings, two test cases were defined as crucial for the federation of the immuneML to be considered valid. These test cases assess if the merging of the models trained in each node is able to produce significant results, as well as the model behavior under an heterogeneous network, with the majority of the nodes used for training not having significant amounts of data.

Following these guidelines, those meetings resulted in the following two test setups:

Convergence test

In this test case, 50 Data Provider Nodes will be created and run. Each node must perform well when running the centralized pipeline, allowing for the most stable test of the model merging.

This process will be repeated several times in order to assess the impact of node quantity in the merging process. For this, different runs will be made, with two different amount of nodes: **5**, **25**, and **50**.

The goal of this test case is to understand if the merging is working and its limitations regarding the amount of nodes used for training. Merged models will be assessed and reported on to analyze their performance.

Heterogeneity test

For the second test case, 50 Data Provider Nodes will be created and run. Some nodes will be created as subsection 3.4.1, with good performance on the centralized pipeline. The remaining nodes will be created with tiny datasets, meaning they will have a very poor performance on the same pipeline.

This process aims to understand the impact of nodes with little value on the final model performance, with the goal of understanding if heterogeneity may present a problem. The test will be ran with a dataset of 50 nodes, 45 of them being *strong* nodes, and the other 5 being *weak* nodes.

3.4.2 Data generation and signal implantation

Following the already implemented prototype settings, the dataset format is selected as *repertoire*. This section will provide an in-depth view over the data generation YAML files and selected properties.

Convergence test

The first dataset collection will consist of fifty nodes, each with 100 repertoires. Each repertoire will contain 600 sequences of length 32. The YAML file in Listing 3.1 was built and run fifty times to generate the different datasets.

```yaml
definitions:
  datasets:
    my_dataset:
      format: RandomRepertoireDataset
      params:
        repertoire_count: 100 # number of repertoires to be generated
        sequence_count_probabilities:
          600: 1 # all repertoires have 600 sequences
        sequence_length_probabilities:
          32: 1 # sequence length is 32
        labels: null

instructions:
  dataset_export: # instruction 1: export the randomly generated dataset
     in AIRR format
    type: DatasetExport
    datasets:
        - my_dataset
    export_formats:
        - AIRR
```

Listing 3.1: Random dataset generation YAML specification file for 100 repertoires.

ImmuneML's random dataset generation only creates the repertoire files. To be able to run a study on the platform, a metadata file per dataset/node is required, as per section 2.3. A python script was developed to create such file, and some assumptions were made: every repertoire belongs to a different subject, and the subjects' identifier is random and incremental in each metadata file.

In section 2.3, signal injection was studied, as how it can simulate immune events. After generating the datasets, the collection must be injected with a signal, and the process is described in the Listing 3.2.

This specification entails the dataset path, which was replaced by each of the nine nodes' paths in each run. The motif to be implanted is a k-mer with a length of 5, *VERYW*. This motif is applied in the *disease* signal, and this signal specifies that the k-mer must be positioned in between the positions 20 and 23, with even probabilities.

The most important tweak aspect of the simulation is the implanting, where the dataset and repertoire implanting rates can be varied. The dataset implanting rate corresponds to the fraction of the repertoires that will have the signal *disease* implanted, in this case, half of the repertoires in each dataset. The repertoire implanting rate translates to the fraction of sequences, in each *signaled* repertoire, that will actually get the k-mer implanted. A repertoire implanting rate of ten percent was chosen in order to help maintain a high model accuracy. For this work's scope is not to evaluate how the model performs on each dataset, but if the models can be merged.

The simulation also creates a new data column in all nodes' metadata files, with the name of the signal: *disease*. This column will read True if the signal was implanted in the corresponding repertoire, and False otherwise.

```yaml
definitions:
  datasets:
    my_dataset:
      format: AIRR
      params:
        metadata_file: ./dataset/metadata.csv
        path: ./dataset/
        result_path: ./dataset/result-immuneml_imported_files/
  motifs:
    kmer:
      instantiation: GappedKmer
      seed: VERYW
  signals:
    disease:
      implanting: HealthySequence
      motifs:
      - kmer
      sequence_position_weights:
        20: 0.25
        21: 0.25
        22: 0.25
        23: 0.25
  simulations:
    sim:
      implanting1:
        dataset_implanting_rate: 0.5
        repertoire_implanting_rate: 0.1
        signals:
        - disease
instructions:
  sim_instruction:
    dataset: my_dataset
    export_formats:
    - AIRR
    simulation: sim
    type: Simulation
```

Listing 3.2: Motif implantation YAML specification file for a repertoire implanting rate of 10%.

These datasets are saved and arranged in a collection of 50 directories with names from *node0* to *node49*, from which the varying number of nodes, mentioned in subsection 3.4.1, can be selected: either 5, 25, or 50. After the generation, the datasets are joined so that results may be evaluated against data from all nodes, and so that a full model can be trained in a centralized way and the results can be compared. For this, all repertoires' files are copied into a single directory, and all metadata files have their content merged into a single file. As each metadata file identifies their subjects with ascending identifiers starting from *1*, on the merged file these entries are edited and, on the final version, contains identifiers ranging from *1* to *5000*. The merged results are stored in a directory called *server*. A repertoire's file unique fields (Table 3.2), and a metadata's file fields (Table 3.3) examples are shown for reference.

sequence_id	cdr3_aa
0	LPISDEACYYLPASMSKWYHHCQYGKIVFFLR
1	FADMSHKQDYQNLMTEEIQNMHHEWQHTVHEE
2	KPMVPDNRRWEKLMVGLEQYSYRGFPDCSRKR
3	LPKAHNTHNQFTPMEIHRTRLLNAQIHGFVWH
4	WLFTVEDDNSGPEEGDKNHRRELDAWEVWVMS
5	YAVNPEGLKTWPLMRFNLAGMDVYASSWRNQG
...	...
595	QCCDWKLGPTFIRSSRDWWTLLTENGYSHGNC
596	LYEWKIGTIAKADKFQILSPFPYRAASIWILL
597	EDCYTMCAARRFYCDYTVAWQKSEPRYIADDF
598	TMLHHEQWLAAFKGEARTGFNQQEHDKWCDDI
599	PVPWLNMHKWGECVCEAVPRADGFAKCEWSWW

Table 3.2: Generated repertoire file example

disease	filename	identifier	subject_id
True	repertoires/a.tsv	a	1
True	repertoires/b.tsv	b	2
True	repertoires/c.tsv	c	3
True	repertoires/d.tsv	d	4
True	repertoires/e.tsv	e	5
...	...	...	...
False	repertoires/v.tsv	v	96
False	repertoires/w.tsv	w	97
False	repertoires/x.tsv	x	98
False	repertoires/y.tsv	y	99
False	repertoires/z.tsv	z	100

Table 3.3: Generated metadata file example

Heterogeneity test

As explained in subsection 3.4.1, this test-case will use the same datasets from the previous test-case for the *strong* nodes, and new *weak* datasets will be generated. Following the same principle, the *weak* datasets will be generated using the YAML file in Listing 3.3

```
definitions:
  datasets:
    my_dataset:
      format: RandomRepertoireDataset
      params:
        repertoire_count: 6 # number of repertoires to be generated
        sequence_count_probabilities:
          600: 1 # all repertoires have 600 sequences
        sequence_length_probabilities:
          32: 1 # sequence length is 32
        labels: null

instructions:
  dataset_export: # instruction 1: export the randomly generated dataset
    in AIRR format
    type: DatasetExport
    datasets:
        - my_dataset
    export_formats:
        - AIRR
```

Listing 3.3: Random dataset generation YAML specification file for single repertoire.

This generation process will be ran several times, as a number of tests is performed. As shown in subsection 3.4.1, 5 datasets of *weak* nodes will be generated, which may then be used to test the pipeline. Tests will be ran the same way as in the Convergence test-case, with a joined model being created, and both centralized and federated analysis being ran and reported on.

For the motif implanting, the same file as the Convergence test-case nodes will be used, meaning all simulated datasets will have the same implanting rates, as well as the same signal properties, such as sequence positioning and motif seed.

3.5 Implementation and results

This section will cover how the Flower framework subsection 2.4.5 was implemented in the immuneML prototype described in section 3.1, the changes that were made to some packages of the prototype in order for it to comply with federated training, and will finally present the results of the Convergence and Heterogeneity test-cases, which were described in subsection 3.4.1.

3.5.1 Flower implementation

In order to integrate the Flower federated framework with the built prototype, some changes were required. The **Client** and **Server** modules were coded into the platform, the **FederatedLogisticRegression** and **FederatedCoefficients** classes were created as wrappers for the immuneML's LogisticRegression ML method, and Coefficients Report, respectively, in order to add, or customize their features. The original **FedAvg** package from Flower, and the **MLTrainer**, and **MLAssessment** from the existing prototype, were lightly modified to adapt to the new requirements, and **MLReport** was created, based on the latter two packages' configurations. For this framework there was also needed an utilitary package **Utils**, which connects various services within the new prototype. This part of the document will focus on the **Client**, the **Server**, and the **Utils** packages, as well as the **FederatedLogisticRegression** wrapper, and the settings used for the **FedAvg** merging strategy. Other packages will not be discussed here as they only required changes in order to work along with these implementations.

Some packages, related to encoding and importing, were re-used from immuneML. Since they were already explained in section 3.2, they will not be covered in this section.

Client

Following the pipeline presented in subsection 3.3.2, the **Client** performs the 9 first steps in the same way the centralized pipeline does. Initially, the **Client** runs the defines the parameters that will be used by the encoder, as seen in Listing 3.4.

```
encoding_arguments = {
    'encoding': 'KmerFrequency',
    'sequence_encoding': 'CONTINUOUS_KMER',
    'reads': 'UNIQUE',
    'sequence_type': 'AMINO_ACID',
    'k': 3
}
```

Listing 3.4: Encoder parameters.

Then, the dataset is imported, and the merge process is ran, which is always necessary, even if only one dataset is imported. The normalization and scaling steps are ran, and finally the dataset is split in training and testing parts. The code for these steps can be seen in Listing 3.5.

```
original_encoded_data = utils.get_encoded_dataset(encoding_arguments,
    node_id)
encoded_datasets = EncodedDatasets(encoding = encoding_arguments['
    encoding'], params = encoding_arguments)
encoded_datasets.add_dataset(original_encoded_data)

encoded_datasets.merge_datasets()

encoded_datasets.normalizationType = NormalizationType('max')
encoded_datasets.scaled_to_zero_mean = False

encoded_datasets.split_normalize(0.2)
```

Listing 3.5: Dataset import, merge, encoding, normalization, and split.

After the datasets are processed, the ML model must be initiated, requiring the maximum number of iterations to be set, and the warm_start flag to be activated. The number of iterations was kept high in order to not limit the model, and have a high probability of a successful convergence. The warm_start flag is used because without it the LinearRegression ML model resets its own coefficients (or weights) before the fitting process, meaning the model would be trained from scratch, instead of training over the dataset received from the **Server**. Regardless of the warm_start flag, this initialization process is happening before any round of actual training, so the model must be initialized with the correct parameter shape, in order to match the other **Clients**, and allow for posterior merging. The model initialization code is detailed in Listing 3.6.

```
new_model = LogisticRegression(
    max_iter=20,
    warm_start=True
)

utils.set_initial_params(new_model)
```

Listing 3.6: Client ML model initialization.

With the datasets processed, and the model initialized, the NodeMLClient class is defined, which extends directly from Flower's Client. This class defines 3 functions, which are exposed to the **Server** during the federated learning cycle, meaning the **Server** can call these functions in order to define the flow of the training process. All the client functions also receive a *config* parameter, which can share important data between the **Server** and the **Client**, primarily the current round of training. The first function is used to retrieve the model parameters from the **Client**, usually after each training round, and is show in Listing 3.7.

```
def get_parameters(self, config):
    return utils.get_model_parameters(self.model)
```

Listing 3.7: Client get_parameters method.

The second function provided by the **Client** is the model fitting function. This function is called by the **Server** for each round of training that the **Client** is selected to participate in. Initially, the method sets the ML model parameters received, meaning that, after each round of training, the merged coefficients are received from the **Server** and set, so that the model will now train over this data, and improve the received model. This is followed by the actual model training instructions, using the training portion of data from the encoded dataset, and finally, the method returns the model's parameters as well as the length of the dataset the **Client** used. The coding details can be seen in Listing 3.8.

```
def fit(self, parameters, config):
    utils.set_model_params(self.model, parameters)

    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        self.model = utils.train_model(node_id, encoded_datasets.
    encoding, encoded_datasets.train_encoded_data, f"/path/to/results/
    rnd_{config['server_round']}", 'log_loss', self.model)
    print(f"Training finished for round {config['server_round']}")

    return utils.get_model_parameters(self.model), len(encoded_datasets.
    train_encoded_data.example_ids), {}
```

Listing 3.8: Client fit method.

Finally, the **Client** exposes an evaluation function, so that the locally trained model may be assessed. This function starts by also setting the coefficients it receives from the **Server** to the model. This function uses the testing part of the encoded dataset to perform an analysis, resulting in the logarithmic loss, and accuracy of the model being returned, along with the amount of samples used for testing. The code for the assessment is present in Listing 3.9.

```
def evaluate(self, parameters, config):
    utils.set_model_params(self.model, parameters)

    ml_assessment = MLAssessment(node_id, encoded_datasets.
   test_encoded_data, self.model, 'auc', f"/path/to/results/rnd_{config
   ['server_round']}", [Metric.LOG_LOSS, Metric.ACCURACY])
    scores = ml_assessment.run()

    return float(log_loss), len(encoded_datasets.test_encoded_data.
   example_ids), {"accuracy": accuracy}
```

Listing 3.9: Client evaluate method.

Every function presented uses some form of reading from, or writing to the actual ML model, and the code for these was made available in the utilities package **Utils**. Furthermore, since the training and assessment process were also activities performed on the models, the decision was made to offload these code bits to the utilities package as well. This decision stems from the reusing of that code by other packages, including the **Server**, avoiding any needless and potentially hazardous code duplication.

Server

like Client, adds strategy, like FedAvg

The **Server** package also follows the first 9 steps of the centralized pipeline in the same way the **Client** does, having the same initialization code seen in Listing 3.4 and Listing 3.5. Since the **Server** never performs a training step itself, the model can be initialized without any parameters, as the fitting is the only process that resets the model's coefficients. The new initialization process is shown in Listing 3.10, along with the utility function that defines the model initial parameters.

```
new_model = LogisticRegression()

utils.set_initial_params(new_model)
```

Listing 3.10: Server ML model initialization.

The next function, presented in Listing 3.11, is an interface for the **Client** to be able to query the **Server** on which round of training is happening, allowing for the possibility of conditioning the pipeline flow based on such value.

```
def fit_round(server_round: int):
    return {"server_round": server_round}
```

Listing 3.11: Server fit_round method.

Finally, the **Server** must be initialized with the training settings, namely the strategy, that is going to be used for the model merging process, and the number of rounds of training, that limits the pipeline to the specified number of executions. The starting method is show in Listing 3.12.

```
fl.server.start_server(
    server_address="0.0.0.0:8080",
    strategy=strategy,
    config=fl.server.ServerConfig(num_rounds=10)
)
```

Listing 3.12: Server start command.

Utils

The utility package **Utils** is a simple interface between the prototype's code and the actual ML model. Since both the **Server** and the **Client** need to perform certain actions on the LogisticRegression model, this package was created in order to group the needed functions, and reduce code duplication. Methods to retrieve and set model parameters are detailed in Listing 3.13 and Listing 3.14.

```
def get_model_parameters(model: LogisticRegression
   ):
    return model.get_parameters()
```

Listing 3.13: Utility to retrieve model parameters.

```
def set_model_params(model: LogisticRegression,
   params: LogRegParams):
    model.set_params(params)
    return model
```

Listing 3.14: Utility to set model parameters.

After running the ML model in the centralized pipeline, the amount of classes and features of the dataset were analyzed and used to create the code for the initial parameters set method, shown in Listing 3.15. This guarantees the heterogeneity of models between all **Clients**, and now, the **Server**.

```
def set_initial_params(model: LogisticRegression):
    n_classes = 2
    n_features = 8000
    classes_ = np.array([i for i in range(n_classes)])

    coef_ = np.zeros((n_classes, n_features))
    intercept_ = np.zeros((n_classes,))

    model.set_initial_params(model._get_ml_model(), classes_, coef_,
   intercept_)
```

Listing 3.15: Utility to set model's initial parameters.

Listing 3.8 also shows the usage of the method *train_model*, which only goal is to verify the presence of an optimization metric before launching the **MLTrainer** runner.

FederatedLogisticRegression wrapper

The **FederatedLogisticRegression** wrapper was built in order to modify, and add methods from the immuneML's LogisticRegression wrapper, which in its turn, wraps around the original sklearn's LogisticRegression. Methods were developed to enable the model parameters retrieval, and editing, and to enable the initial parameters to be set. The getter and setter methods, described in Listing 3.16 and Listing 3.17, respectively, simply return or overwrites the model's *coef_* and *intercept_* variables, which is the data the model needs in order to be shared and fully functional when restored.

```
def get_parameters(self):
    params = [
        self.model.coef_,
        self.model.intercept_,
    ]
    return params
```

Listing 3.16: Federated Logistic Regression getter.

```
def set_params(self, params):
    self.model.coef_ = params[0]
    self.model.intercept_ = params[1]
```

Listing 3.17: Federated Logistic Regression setter.

As for the initial parameters, the *classes_* variable must also be set, since no action can be performed on the ML model without it. The code to this method is represented in Listing 3.18.

```
def set_initial_params(self, model, classes_,
   coef_, intercept_):
    self.model = model
    self.model.classes_ = classes_
    self.model.coef_ = coef_
    self.model.intercept_ = intercept_
```

Listing 3.18: Federated Logistic Regression initial parameter setter.

FedAvg

The **FedAvg**, or Federated Average, was chosen to aggregate the model. In order to initialize this strategy, some parameters must be defined. First, the *min_available_clients* is set, which defines how many **Clients** must be actively connected to the **Server**, and the *min_fit_clients*, and *min_evaluate_clients*, which define the minimum amount of **Clients** that will be pooled from all the connected ones, in order to train and evaluate the model, respectively. The entries *fraction_fit* and *fraction_evaluate* also represent the amount of **Clients** used in training and evaluation, respectively, but should this percentage result in a number of clients below the minimum set in *min_fit_clients*, or *min_evaluate_clients*, it will be ignored and the minimum number will prevail. This configuration is detailed in Listing 3.19, and since this work is based on a fixed amount of nodes, all entries equate to the number of **Clients** being used in each test.

```
strategy = fl.server.strategy.FedAvg(
    min_available_clients=50,
    min_fit_clients=50,
    min_evaluate_clients=50,
    fraction_fit=1,
    fraction_evaluate=1,
)
```

Listing 3.19: FedAvg strategy parameters.

3.5.2 Convergence test

In this section we will present the model training loss and accuracy, for each of the tested scenarios, along with the reports from the centralized run on the whole dataset, and the federated run, for comparison.

Initially, as the nodes are similar between them, we perform a centralized run on a single node and present the same metrics for it, as well as the generated report in Figure 3.3:

- **log_loss** - 0.4510971034124941
- **accuracy** - 1.0

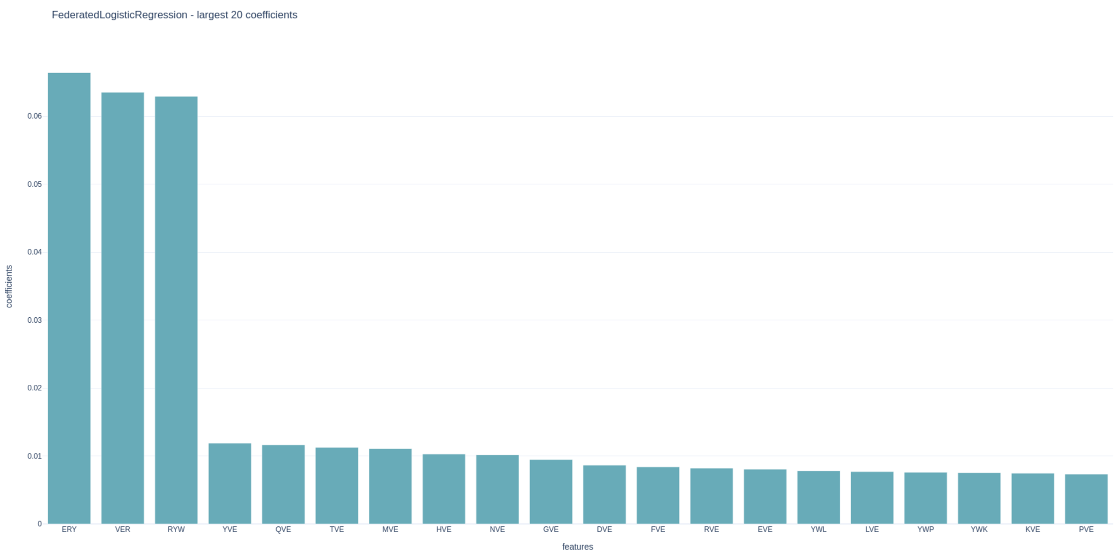

Figure 3.3: Coefficient report on single node centralized model.

5 nodes

- **Centralized**
  - **log_loss** - 0.18608316642434025
  - **accuracy** - 1.0
- **Federated**
  - **log_loss** - 0.398587337279971
  - **accuracy** - 1.0

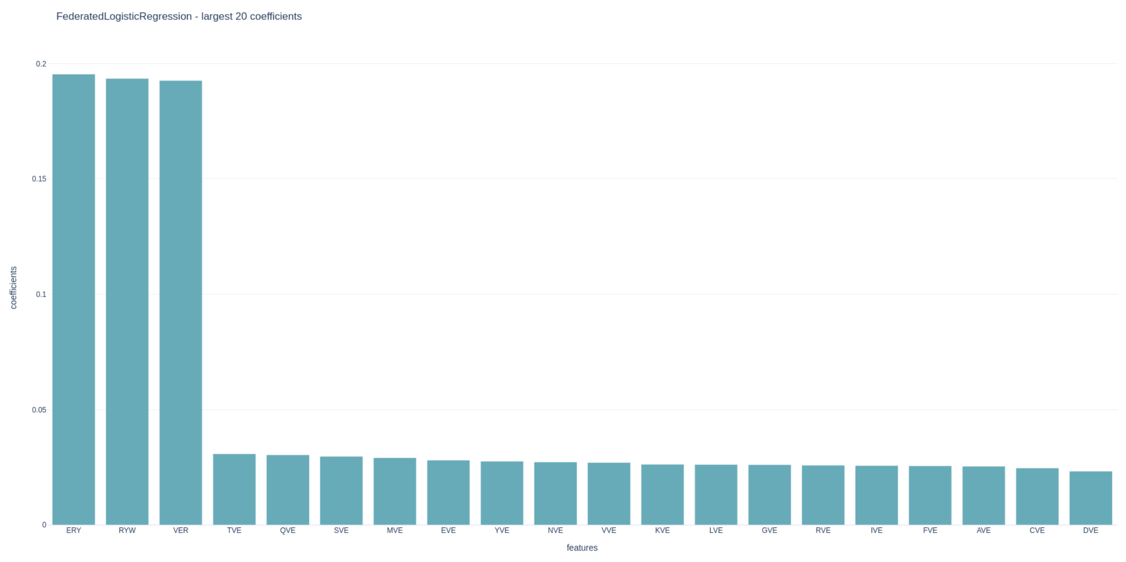

Figure 3.4: Coefficient report on the 5 node centralized model.

25 nodes

- **Centralized**
  - **log_loss** - 0.03238872034032445
  - **accuracy** - 1.0
- **Federated**
  - **log_loss** - 0.37750687996030513
  - **accuracy** - 1.0

50 nodes

- **Centralized**
  - **log_loss** - 0.01672917265554102
  - **accuracy** - 1.0
- **Federated**
  - **log_loss** - 0.3205385183804359
  - **accuracy** - 1.0

3.5.3 Heterogeneity test

This section will be similar to subsection 3.5.2, but this time will include the *weaker* nodes. The results will be presented in a similar manner, and, for comparison, the centralized pipeline was ran against a single *weak* node. As was expected, a dataset with only 6 entries will not perform very well, and immuneML ends up returning a *not_computed* error for the model losses, so no analysis is performed on a single *weak* node.

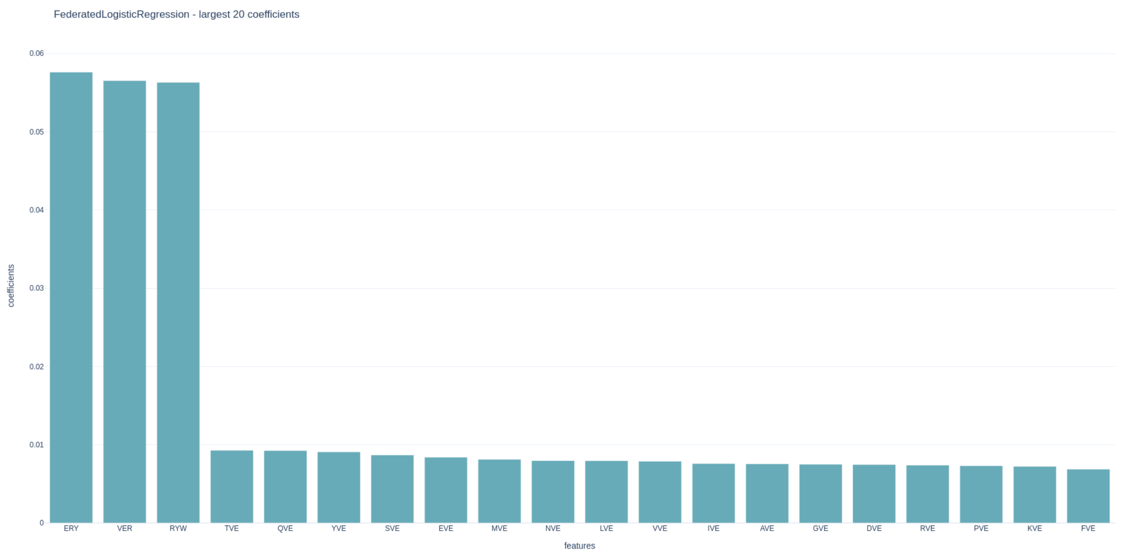

Figure 3.5: Coefficient report on the 5 node federated model.

5 weak + 45 strong nodes

- **Centralized**
  - **log_loss** - 0.017744661679538774
  - **accuracy** - 1.0
- **Federated**
  - **log_loss** - 0.3770383377311905
  - **accuracy** - 1.0

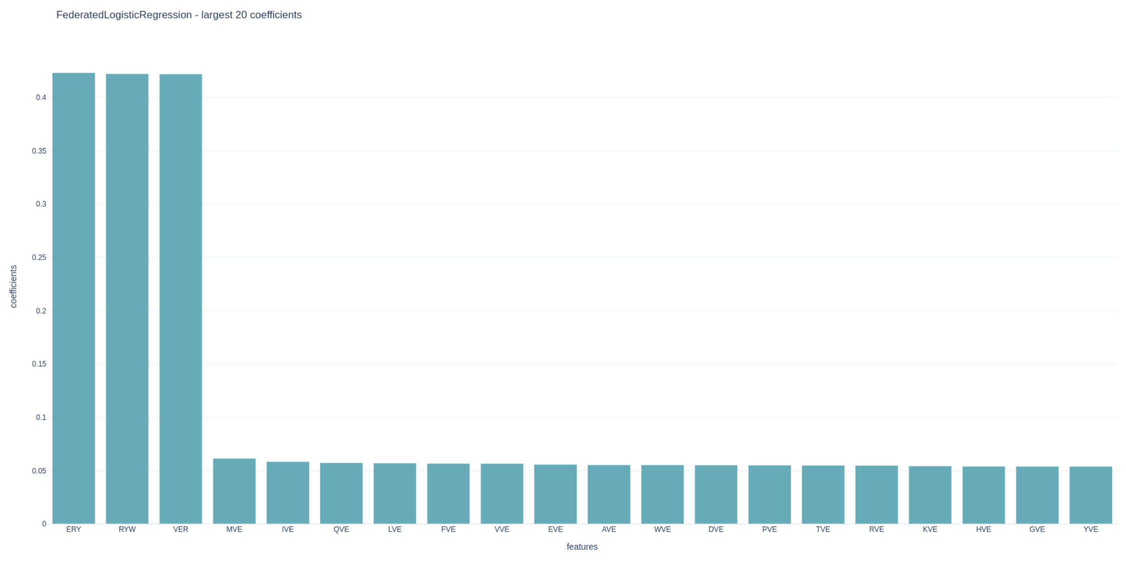

Figure 3.6: Coefficient report on the 25 node centralized model.

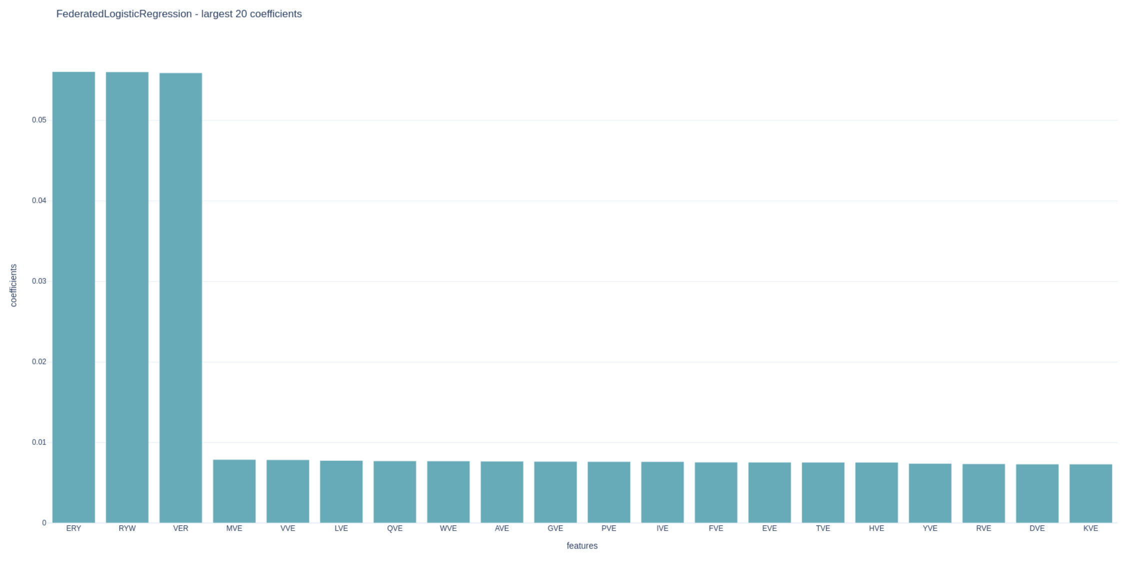

Figure 3.7: Coefficient report on the 25 node federated model.

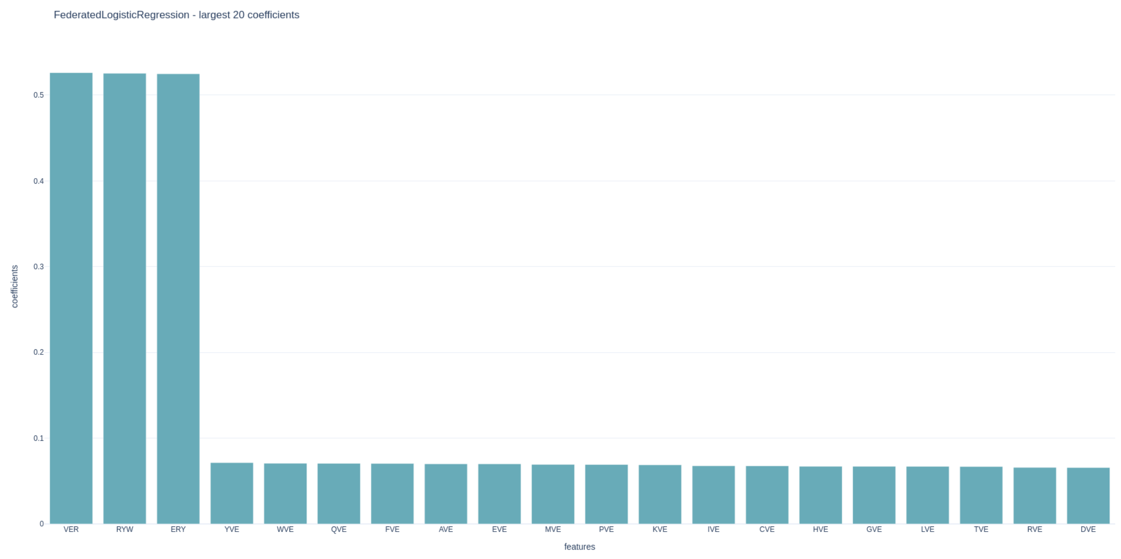

Figure 3.8: Coefficient report on the 50 node centralized model.

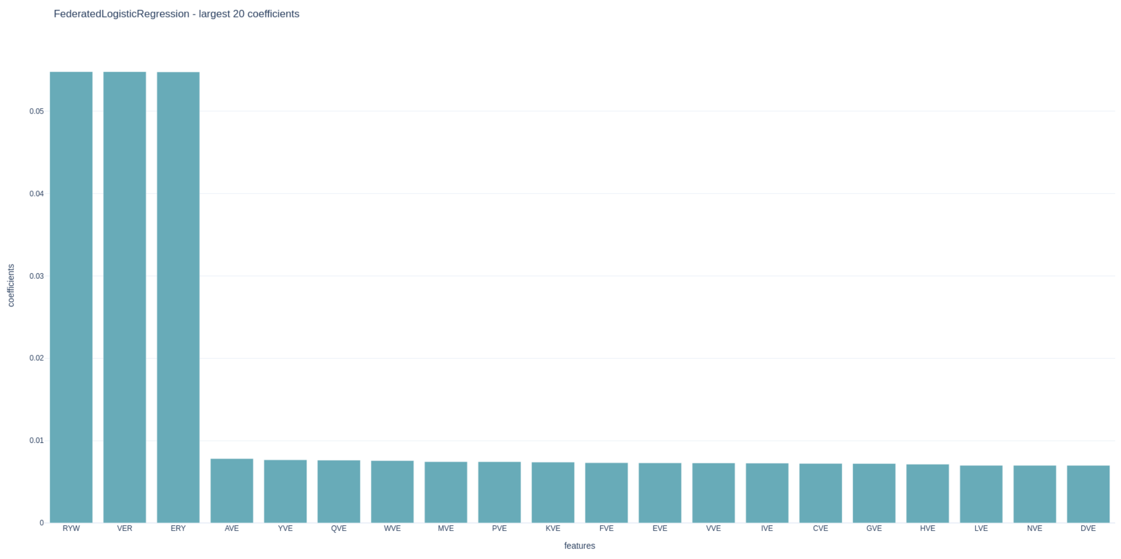

Figure 3.9: Coefficient report on the 50 node federated model.

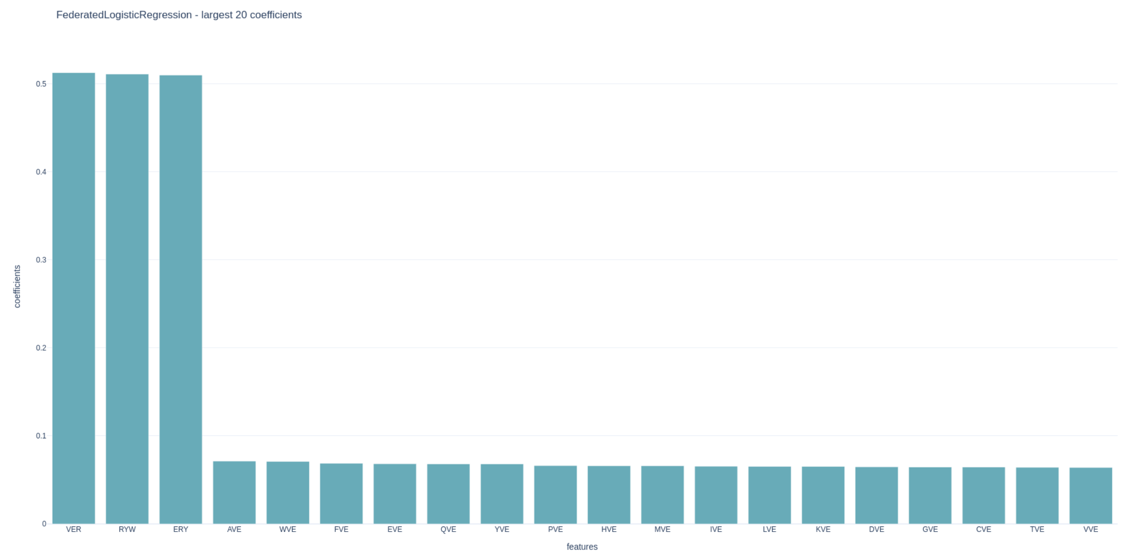

Figure 3.10: Coefficient report on the 5/45 node centralized model.

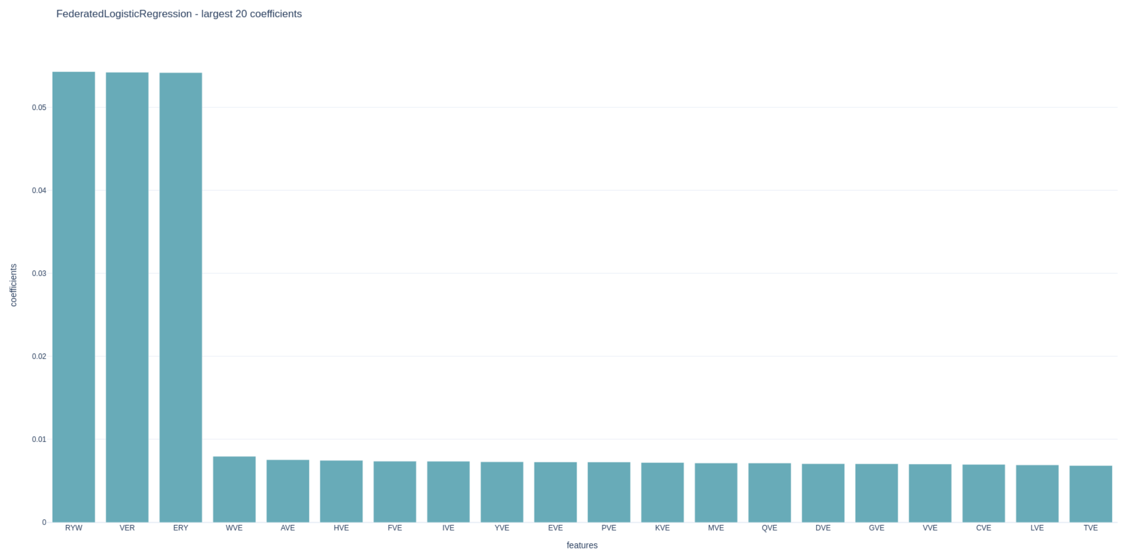

Figure 3.11: Coefficient report on the 5/45 node federated model.

Chapter 4

Conclusion

4.1 Conclusions

The main goal of this dissertation is to improve the decentralized training prototype by enabling more analysis, adding training ability to the Data Provider Nodes and improving the user interface. As literature shows, decentralization is a good privacy-preserving measure, although it must be tailored to each specific analysis type, and the usage of such frameworks allows for more reproducible results.

While working for TPPROJIA and IAAPLIA courses, the context, motivation and main goals for this work have been detailed. Regarding the research of the state of the art on ML reproducibility, Federated Learning, and different techniques and tools were reported. Finally an experimentation guide for an initial prototype of the solution was fully explained.

The federated prototype enables immunotherapy studies to be performed across several devices, without ever releasing the original information from the datasets. By enabling each data owning node to perform the ML training locally, we take away some of the responsibility and legal obstacles to large scale medical ML studies, possibly improving future research.

In order to have the prototype validated, meetings with immuneML members were held, where there was very good feedback and interest in the platform. Those meetings not only served to move forward in the development phase, but also to understand the implications and possible future work for the prototype.

Furthermore, we can understand that the convergence of the models in a federated learning environment is possible, whether the total amount of nodes is small or big, and even if there are some nodes within the study which lack the information quantity, compared to the other ones. This means that the prototype could potentially be used for research in both large-scale and small-scale studies and research operations. Since the convergence is not highly affected by the presence of *weak* nodes, it is also safe to assume that the selection of nodes and available datasets may not follow as strict rules as in other ML scenarios.

4.2 Future Work

There are a lot of features that could potentially improve, or at least help understand better, what can and cannot be done with the developed prototype. After some meetings with the immuneML team, and following the research done throughout the document, some ideas were gathered as to what could be done next:

- There should be more strategies tested, as to gain an understanding on the different applications of each one;
- There should be tests done with more and bigger nodes to understand if scalability is an issue, although much more powerful systems will be needed to test and simulate this;
- The data heterogeneity should be tested by running examples with very different data configurations between them;
- The integration with a Web API, and a posterior front-end development should be considered in order to improve the usability of the platform.

Bibliography

Bagaev, Dmitry V et al. (Jan. 2020). "VDJdb in 2019: database extension, new analysis infrastructure and a T-cell receptor motif compendium". In: *Nucleic Acids Research* 48.D1, pp. D1057–D1062. issn: 0305-1048. doi: 10.1093/nar/gkz874. url: https://doi.org/10.1093/nar/gkz874 (visited on 10/23/2022).

Banna, Vishnu et al. (July 2021). "An Experience Report on Machine Learning Reproducibility: Guidance for Practitioners and TensorFlow Model Garden Contributors". In: *arXiv:2107.00821 [cs]*. arXiv: 2107.00821. url: http://arxiv.org/abs/2107.00821 (visited on 01/21/2022).

Beam, Andrew L., Arjun K. Manrai, and Marzyeh Ghassemi (Jan. 2020). "Challenges to the Reproducibility of Machine Learning Models in Health Care". In: *JAMA* 323.4, pp. 305–306. issn: 0098-7484. doi: 10.1001/jama.2019.20866. url: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7335677/ (visited on 09/15/2022).

Bernal, Daniel García (n.d.). "Decentralizing Large-Scale Natural Language Processing With Federated Learning". en. In: (), p. 84.

Beutel, Daniel J. et al. (Mar. 2022). *Flower: A Friendly Federated Learning Research Framework*. arXiv:2007.14390 [cs, stat]. doi: 10.48550/arXiv.2007.14390. url: http://arxiv.org/abs/2007.14390 (visited on 10/11/2022).

Bolotin, Dmitriy A. et al. (May 2015). "MiXCR: software for comprehensive adaptive immunity profiling". en. In: *Nature Methods* 12.5. Number: 5 Publisher: Nature Publishing Group, pp. 380–381. issn: 1548-7105. doi: 10.1038/nmeth.3364. url: https://www.nature.com/articles/nmeth.3364 (visited on 10/23/2022).

Bonawitz, Keith et al. (Apr. 2019). "Towards Federated Learning at Scale: System Design". en. In: *Proceedings of Machine Learning and Systems* 1, pp. 374–388. url: https://proceedings.mlsys.org/paper/2019/hash/bd686fd640be98efaae0091fa301e613-Abstract.html (visited on 10/24/2022).

Breden, Felix et al. (2017). "Reproducibility and Reuse of Adaptive Immune Receptor Repertoire Data". In: *Frontiers in Immunology* 8. issn: 1664-3224. url: https://www.frontiersin.org/article/10.3389/fimmu.2017.01418 (visited on 01/25/2022).

Choudhury, Olivia et al. (Feb. 2020). "Differential Privacy-enabled Federated Learning for Sensitive Health Data". In: *arXiv:1910.02578 [cs]*. arXiv: 1910.02578. url: http://arxiv.org/abs/1910.02578 (visited on 01/24/2022).

Christley, Scott et al. (2020). "The ADC API: A Web API for the Programmatic Query of the AIRR Data Commons". In: *Frontiers in Big Data* 3. issn: 2624-909X. url: https://www.frontiersin.org/articles/10.3389/fdata.2020.00022 (visited on 10/23/2022).

Fallah, Alireza, Aryan Mokhtari, and Asuman Ozdaglar (n.d.). "Personalized Federated Learning with Theoretical Guarantees: A Model-Agnostic Meta-Learning Approach". en. In: (), p. 12.

Gad, Ahmed (Apr. 2020). *Introduction to Federated Learning*. Tech. rep. doi: 10.13140/RG.2.2.34366.51521.

Goncharov, Mikhail et al. (Sept. 2022). "VDJdb in the pandemic era: a compendium of T cell receptors specific for SARS-CoV-2". en. In: *Nature Methods* 19.9. Number: 9 Publisher:

Nature Publishing Group, pp. 1017–1019. issn: 1548-7105. doi: 10.1038/s41592-022-01578-0. url: https://www.nature.com/articles/s41592-022-01578-0 (visited on 10/23/2022).

Gundersen, Odd Erik, Saeid Shamsaliei, and Richard Juul Isdahl (Jan. 2022). "Do machine learning platforms provide out-of-the-box reproducibility?" en. In: *Future Generation Computer Systems* 126, pp. 34–47. issn: 0167-739X. doi: 10.1016/j.future.2021.06.014. url: https://www.sciencedirect.com/science/article/pii/S0167739X21002090 (visited on 01/21/2022).

Heil, Benjamin J. et al. (Oct. 2021). "Reproducibility standards for machine learning in the life sciences". en. In: *Nature Methods* 18.10. Bandiera_abtest: a Cg_type: Nature Research Journals Number: 10 Primary_atype: Comments & Opinion Publisher: Nature Publishing Group Subject_term: Data publication and archiving;Machine learning;Standards Subject_term_id: data-publication-and-archiving;machine-learning;standards, pp. 1132–1135. issn: 1548-7105. doi: 10.1038/s41592-021-01256-7. url: https://www.nature.com/articles/s41592-021-01256-7 (visited on 01/21/2022).

Hutson, Matthew (Feb. 2018). "Artificial intelligence faces reproducibility crisis". In: *Science (New York, N.Y.)* 359, pp. 725–726. doi: 10.1126/science.359.6377.725.

immuneML documentation (2022). url: https://docs.immuneml.uio.no/latest/index.html (visited on 01/28/2022).

Jabbari, Parnian and Nima Rezaei (July 2019). "Artificial intelligence and immunotherapy". en. In: *Expert Review of Clinical Immunology* 15.7, pp. 689–691. issn: 1744-666X, 1744-8409. doi: 10.1080/1744666X.2019.1623670. url: https://www.tandfonline.com/doi/full/10.1080/1744666X.2019.1623670 (visited on 10/09/2022).

Khan, Latif U. et al. (Sept. 2020). *Federated Learning for Edge Networks: Resource Optimization and Incentive Mechanism*. arXiv:1911.05642 [cs]. doi: 10.48550/arXiv.1911.05642. url: http://arxiv.org/abs/1911.05642 (visited on 10/10/2022).

Li, Tian, Anit Kumar Sahu, Ameet Talwalkar, et al. (May 2020). "Federated Learning: Challenges, Methods, and Future Directions". In: *IEEE Signal Processing Magazine* 37.3. Conference Name: IEEE Signal Processing Magazine, pp. 50–60. issn: 1558-0792. doi: 10.1109/MSP.2020.2975749.

Li, Tian, Anit Kumar Sahu, Manzil Zaheer, et al. (Apr. 2020). *Federated Optimization in Heterogeneous Networks*. arXiv:1812.06127 [cs, stat]. doi: 10.48550/arXiv.1812.06127. url: http://arxiv.org/abs/1812.06127 (visited on 10/10/2022).

Li, Tian, Maziar Sanjabi, et al. (Feb. 2020). *Fair Resource Allocation in Federated Learning*. arXiv:1905.10497 [cs, stat]. doi: 10.48550/arXiv.1905.10497. url: http://arxiv.org/abs/1905.10497 (visited on 10/10/2022).

Mammen, Priyanka Mary (Jan. 2021). "Federated Learning: Opportunities and Challenges". In: *arXiv:2101.05428 [cs]*. arXiv: 2101.05428. url: http://arxiv.org/abs/2101.05428 (visited on 01/24/2022).

McDermott, Matthew B. A. et al. (July 2019). "Reproducibility in Machine Learning for Health". In: *arXiv:1907.01463 [cs, stat]*. arXiv: 1907.01463. url: http://arxiv.org/abs/1907.01463 (visited on 01/21/2022).

McMahan, H. Brendan et al. (Feb. 2017). *Communication-Efficient Learning of Deep Networks from Decentralized Data*. arXiv:1602.05629 [cs]. doi: 10.48550/arXiv.1602.05629. url: http://arxiv.org/abs/1602.05629 (visited on 09/12/2022).

Mining adaptive immune receptor repertoires for biological and clinical information using machine learning - ScienceDirect (2022). url: https://www.sciencedirect.com/science/article/pii/S2452310020300524 (visited on 10/09/2022).

Onose, Ejiro (July 2021). *How to Solve Reproducibility in ML*. en-US. url: https://neptune.ai/blog/how-to-solve-reproducibility-in-ml (visited on 09/15/2022).

Pavlović, Milena et al. (Mar. 2021). *immuneML: an ecosystem for machine learning analysis of adaptive immune receptor repertoires*. en. Tech. rep. Section: New Results Type: article. bioRxiv, p. 2021.03.08.433891. doi: 10.1101/2021.03.08.433891. url: https://www.biorxiv.org/content/10.1101/2021.03.08.433891v3 (visited on 01/27/2022).

Pineau, Joelle et al. (Dec. 2020). "Improving Reproducibility in Machine Learning Research (A Report from the NeurIPS 2019 Reproducibility Program)". In: *arXiv:2003.12206 [cs, stat]*. arXiv: 2003.12206. url: http://arxiv.org/abs/2003.12206 (visited on 01/21/2022).

Platform to Integrate Distributed Repositories of AIRR-seq Data (2022). en-US. url: https://www.ireceptor-plus.com/the-platform/overview/ (visited on 01/25/2022).

Promote Integration of Large AIRR-seq Data (2022). en-US. url: https://www.ireceptor-plus.com/about-us/overview/ (visited on 01/25/2022).

Rieke, Nicola et al. (Sept. 2020). "The future of digital health with federated learning". en. In: *npj Digital Medicine* 3.1. Bandiera_abtest: a Cc_license_type: cc_by Cg_type: Nature Research Journals Number: 1 Primary_atype: Reviews Publisher: Nature Publishing Group Subject_term: Medical imaging;Medical research Subject_term_id: medical-imaging;medical-research, pp. 1–7. issn: 2398-6352. doi: 10.1038/s41746-020-00323-1. url: https://www.nature.com/articles/s41746-020-00323-1 (visited on 01/24/2022).

Rodríguez-Barroso, Nuria et al. (Dec. 2020). "Federated Learning and Differential Privacy: Software tools analysis, the Sherpa.ai FL framework and methodological guidelines for preserving data privacy". en. In: *Information Fusion* 64, pp. 270–292. issn: 1566-2535. doi: 10.1016/j.inffus.2020.07.009. url: https://www.sciencedirect.com/science/article/pii/S1566253520303213 (visited on 10/24/2022).

Rosenfeld, Aaron M. et al. (2018). "ImmuneDB, a Novel Tool for the Analysis, Storage, and Dissemination of Immune Repertoire Sequencing Data". In: *Frontiers in Immunology* 9. issn: 1664-3224. url: https://www.frontiersin.org/articles/10.3389/fimmu.2018.02107 (visited on 10/23/2022).

Rubelt, Florian et al. (Nov. 2017). "Adaptive Immune Receptor Repertoire Community recommendations for sharing immune-repertoire sequencing data". In: *Nature immunology* 18.12, pp. 1274–1278. issn: 1529-2908. doi: 10.1038/ni.3873. url: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5790180/ (visited on 01/25/2022).

Ryffel, Theo et al. (Nov. 2018). *A generic framework for privacy preserving deep learning*. arXiv:1811.04017 [cs, stat]. doi: 10.48550/arXiv.1811.04017. url: http://arxiv.org/abs/1811.04017 (visited on 10/11/2022).

Samuel, Sheeba, Frank Löffler, and Birgitta König-Ries (June 2020). "Machine Learning Pipelines: Provenance, Reproducibility and FAIR Data Principles". In: *arXiv:2006.12117 [cs, stat]*. arXiv: 2006.12117. url: http://arxiv.org/abs/2006.12117 (visited on 01/21/2022).

Shugay, Mikhail et al. (Jan. 2018). "VDJdb: a curated database of T-cell receptor sequences with known antigen specificity". In: *Nucleic Acids Research* 46.D1, pp. D419–D427. issn: 0305-1048. doi: 10.1093/nar/gkx760. url: https://doi.org/10.1093/nar/gkx760 (visited on 10/23/2022).

TensorFlow Federated (2022). en. url: https://www.tensorflow.org/federated (visited on 10/11/2022).

Vander Heiden, Jason Anthony et al. (2018). "AIRR Community Standardized Representations for Annotated Immune Repertoires". In: *Frontiers in Immunology* 9. issn: 1664-3224.

url: https://www.frontiersin.org/articles/10.3389/fimmu.2018.02206 (visited on 10/23/2022).

Wilkinson, Mark D. et al. (Mar. 2016). “The FAIR Guiding Principles for scientific data management and stewardship”. en. In: *Scientific Data* 3.1. Number: 1 Publisher: Nature Publishing Group, p. 160018. issn: 2052-4463. doi: 10.1038/sdata.2016.18. url: https://www.nature.com/articles/sdata201618 (visited on 09/17/2022).

Yang, Qiang et al. (Jan. 2019). “Federated Machine Learning: Concept and Applications”. In: *ACM Transactions on Intelligent Systems and Technology* 10.2, 12:1–12:19. issn: 2157-6904. doi: 10.1145/3298981. url: https://doi.org/10.1145/3298981 (visited on 10/24/2022).

Yu, Han et al. (July 2020). “A Sustainable Incentive Scheme for Federated Learning”. In: *IEEE Intelligent Systems* 35.4. Conference Name: IEEE Intelligent Systems, pp. 58–69. issn: 1941-1294. doi: 10.1109/MIS.2020.2987774.

Yu, Tianlong et al. (Apr. 2020). “Learning Context-Aware Policies from Multiple Smart Homes via Federated Multi-Task Learning”. en. In: *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*. Sydney, Australia: IEEE, pp. 104–115. isbn: 978-1-72816-602-5. doi: 10.1109/IoTDI49375.2020.00017. url: https://ieeexplore.ieee.org/document/9097597/ (visited on 10/10/2022).